SUBSTRATE SUPPORT FOR PEER-TO-PEER APPLICATIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Vivek Vishnumurthy

January 2009

SUBSTRATE SUPPORT FOR PEER-TO-PEER APPLICATIONS

Vivek Vishnumurthy, Ph.D.

Cornell University 2009

Peer-to-peer (P2P) applications in general, and unstructured applications in particular, have been very popular in the recent past. In this thesis, we identify common problems encountered in the course of developing different diverse peer-to-peer applications, and propose solutions to them. The broad problems we study are, namely, (i) load-balancing in heterogeneous unstructured P2P networks, where capacities to support load differ among the different members of the network, and application load is to be distributed in accordance with members' capacities to support it, and (ii) "extreme" nearest neighbor discovery in P2P networks, where the intent is to discover the latency-wise nearest peer in a P2P network, even when the nearest peer is in the same extended LAN or campus network. The goal of this thesis is to come up with a powerful set of basic mechanisms that the developers of many different P2P applications can reuse, rather than having to repeatedly reinvent the same solutions.

We examine two main causes of load in unstructured networks: the underlying random graph, and the process of random node selection, where random peers are periodically picked to handle application load. We extensively evaluate different approaches to do heterogeneous random graph construction and peer selection, and identify our *Swaplinks* algorithm as the best approach. Swaplinks builds robust graphs where node degrees are close to their desired degrees, provides a good base to perform random peer selection, and is virtually free of tuning knobs, making it very practical to deploy.

In our study of the extreme-nearest neighbor discovery problem, we identify and demonstrate a condition we call the *clustering condition* caused by the Internet last-hop architecture – under the clustering condition, many different peers are located at about the same latency from one another, making it expensive for previous nearest-peer solutions to correctly find the nearest peer. We propose different solutions to overcome this problem, and show, using preliminary evaluations, that one of them is very attractive.

BIOGRAPHICAL SKETCH

Vivek was born in the city of Shimoga in Karnataka state in India, and was brought up in Bangalore, Karnataka, India. He got his Bachelor of Technology (B.Tech.) degree from the Indian Institute of Technology at Madras (IIT-Madras) in 2002. He has been in Cornell University from August 2002 to August 2008. He received an M.S. degree from Cornell in 2007 and is expecting the Ph.D. degree to be conferred in January 2009. He will be joining MokaFive, a start-up based on desktop virtualization, as a Software Engineer in September 2008.

ACKNOWLEDGEMENTS

Firstly, I would like to convey my gratitude to my advisor Paul Francis for his guidance throughout my Ph.D. research. His insistence on simplicity and practicality in the course of research is something I hope to retain over my future career. I would also like to thank my committee members Jon Kleinberg and David Shmoys, and Emin Gün Sirer, Robbert van Renesse, and Eva Tardos for their help and comments at various points of my research.

I am grateful to Sergio Gelato, Daniel Kartch, Steve Holland, Aleksey Nogin, Andrew Myers, and Nate Nystrom for developing the Latex template this thesis is based on.

Many thanks to the people who laid out the plans of beautiful Cornell University and Ithaca – towards the end of my PhD I was even enjoying the cold winters here!

I have been lucky to have friendly and helpful officemates throughout my PhD. Heartfelt thanks to Ashwin, Oren, Liviu, Parvati, and Panda.

Many thanks to Saikat, Hitesh, Bernard, Dan, Alan, Oliver, Jed, and Kevin in the Systems Lab for all the fun technical discussions and non-technical interactions over the years.

I have been extremely fortunate to have an awesome and caring set of friends – thanks to Animashree, Ashwin, Lavanya, Mahesh, Muthu, Pappu, Parvathinathan, Parvati, Prakash Linga, Smita, Surabhi, and Vidhyashankar, without whom my stay in Cornell and Ithaca would have been far less enjoyable. Also, thanks to Lavanya, Ashwin, and Parvati for making sure (to the extent possible!) that I was actually working on my thesis when I was supposed to!

Finally, I will forever be indebted to my parents and my sister for having instilled into me the value of education and hard work, for their constant and

unwavering support, and for always encouraging me to strive for the best.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Peer-to-peer applications have been immensely popular for much of the last decade, and have accounted for a major portion of the total Internet traffic in this period. While there is no precise definition of the term peer-to-peer, for the purposes of this thesis, we consider peer-to-peer systems to be distributed systems where all responsibilities involved in running the system are distributed among different participants in the system. Peer-to-peer systems of this kind are generally very robust since they have no single points of failure. In this thesis, we refer to the members of a P2P system as "nodes" or "peers" in the system.

The first popular system labeled as peer-to-peer was the Napster file-sharing system that helped users trade music files: Napster in fact used centralized servers to host the *index* of which users had which files, so is not a "pure" peer-to-peer system according to the above definition [89]. However, the actual exchange of files in Napster was fully peer-to-peer, i.e., from the peer providing the file to the one downloading it. Napster was followed by other file-sharing applications like Gnutella, e-Donkey, Morpheus, Kazaa, BitTorrent, etc.

While peer-to-peer systems became visible with the file-sharing systems, peer-to-peer principles have been used in the Internet for a much longer time, e.g., in Usenet. More recently, peer-to-peer approaches have been used in:

- end-host based multicast (e.g., Yoid [26], Narada [45], Scribe [88], etc.),
- in distributed hash tables (DHTs) (e.g., Chord [99], CAN [80], Pastry [86], Kademlia [65], and Tapestry [42]) to support various applications like content distribution networks (CoralCDN [29]), file-sharing networks (Kad,

BitTorrent without trackers in clients like Azureues and μTorrent), publish/subscribe and multicast systems (scribe [88], bayeux [115], CAN multicast [81]), and distributed file systems (Past [87], CFS [20], Shark [2]) [1],

- Internet telephony (Skype),
- more recently, video streaming applications like TVUPLayer, TVAnts, and PPLive,
- enterprise server-clusters for scalability and availability (Amazon's Dynamo service [21]), and others.

Measurement studies indicate that P2P traffic accounts for a significant fraction of the total traffic exchanged over the Internet [95, 23, 46, 110] [2]. The health of P2P applications thus has a strong effect on the health of the Internet, so there is a lot to be gained from ensuring that the mechanisms used by P2P applications are sound and efficient. This thesis is a step in that direction.

1.1 Contributions of this thesis

In this thesis, we make the observation that developing and successfully operating different large P2P systems often requires solving similar problems – this thesis isolates a few such general problems across diverse P2P systems, and proposes solutions to them. The intent is to develop a single set of mechanisms that can be reused by the developers of diverse applications.

[1] Coral [29] and Shark [2] are based on a "distributed sloppy hash table (DSHT)" abstraction [29], a variant of the DHT abstraction.

[2] the fraction of P2P traffic is about 37% according to Ellacoya [23], between 49% and 83% in various geographic regions according to Ipoque [46], and 60% according to CacheLogic [110]

The first such common issue is the broad problem of load-balancing: each member of a P2P system needs to support some application-based load as a result of being part of the P2P system. Ideally, it should be the case that the load placed on each node is in accordance with its capacity to support the load. An important point to note here is that different nodes might drastically differ in their capacity to bear the application-based load; we term this *heterogeneity* in node-capacities. The problem of load-balancing becomes harder to solve when there is wide heterogeneity in the capacities.

In this thesis, we specifically examine the problem of load-balancing in *unstructured* P2P systems. Peers in all P2P systems maintain connections to a few other peers, called their neighbors, in order to remain connected in spite of node churn and network disruptions. In unstructured P2P systems, there are no restrictions on which nodes can be neighbors – the only restriction in place generally is in terms of the average number of neighbors a given node can maintain. [3] The neighborhood graph in this case can be modeled simply as a random graph. Unstructured P2P applications have been very popular and have dominated usage throughout the history of P2P systems. Examples of unstructured applications include the Gnutella, Kazaa, and BitTorrent file-sharing systems, P2P multicast systems like Chunkyspread [104] and Chainsaw [72], and others.

There could be several "drivers" of application load in unstructured P2P systems: In many settings like unstructured multicast (Chunkyspread [104], Chainsaw [72]) and file-sharing (Gnutella), the higher the degree of a node in the underlying random graph is, the greater is the application load seen by the node. Thus the underlying random graph itself is a driver of the application load. An-

[3] This is in contrast to emphstructured P2P systems, where there are restrictions on which nodes can have neighbor connections between them. Chord [99], CAN [80], Pastry [86], and Tapestry [42] are a few example structured P2P systems.

other important driver is the process of "random selection": In file-sharing protocols like Gnutella, multicast protocols like Yoid [26] and Chunkyspread [104], communication infrastructure like Stunt [37], proximity-prediction systems like Vivaldi [19], whenever a node is to be found to handle load, for instance to look for a file, the mechanism used is to essentially find a random node from the population. In this thesis, we devise and compare multiple solutions to construct random graphs and perform random selection such that the resultant load during both of these processes is proportional to node-capacities. From among these approaches, we identify the *Swaplinks* algorithm as the most attractive unstructured load-balancing mechanism (Chapter 3). We also compare the Swaplinks mechanism with an adapted load-balancing approach based on structured P2P principles in Chapter 4, and conclude that the Swaplinks method is superior when there is high network churn, and is easier to configure and deploy.

The second problem addressed in this thesis is that of discovering the latency-wise nearest peer, especially when the nearest peer is in the same extended LAN or campus network. In P2P settings in general, it is advantageous to communicate with peers that are close-by in terms of latency, rather than those faraway. The advantages are most apparent in the improved user-perceived experience when using latency-sensitive applications like first-person shooter games, and voice/video conferencing. Peers that have small latencies to each other also tend to have higher bandwidths to each other, which leads to better performance as well. In fact, many solutions have already been proposed to solve the problem of find the nearest peer.

In Chapter 5, we identify the *clustering condition* that arises as a side-effect

of the way the Internet last hop is laid out. Under the clustering condition, many different peers end up being at about the same latency from one another, causing existing approaches to fail in correctly finding the nearest peer. We demonstrate the existence of the clustering condition through an extensive measurement over a real P2P population, and a set of DNS servers, and show using analysis and simulations that existing strategies suffer under the clustering condition. We also propose different solutions to overcome this problem, and show, using preliminary evaluations, that one of them is very attractive.

While we focus on the problems of load-balancing and finding the nearest peer in this thesis, these are not the only problems common to different P2P systems. The following is a brief discussion of common problems in P2P systems that we do *not* tackle in this thesis.

Security: Many P2P systems (and approaches presented in this thesis) implicitly trust all participants to correctly follow the specified protocol – this is often a critical assumption in that the system as a whole would break down if this were not the case. For instance, P2P systems require the integrity of the data being exchanged or the computations being performed by peers. They assume that when needed, peers will contribute resources (e.g., bandwidth, CPU, etc.) to sustain the functioning of the system, i.e., that there are no "free-loaders" in the system. In addition, since many P2P systems are pretty open in admitting peers, they are often vulnerable to malicious attacks intended to disrupt the working of the system. A related concern is of privacy: Users might wish to remain anonymous while still participating actively in the P2P system. Approaches in BitTorrent [9], OpenDHT [84], Karma [105], Freenet [16], Onion Routing [36] handle some of these concerns.

"Memory" Tracking: The load-balancing mechanisms we propose in this thesis are specifically targeted toward "memoryless" unstructured P2P applications – we assume an environment where requests that cause load are to be handled by randomly selected nodes, with no regard to the previous history of the specific nodes that handled specific queries. Many structured P2P applications on the other hand need to map a given request or query to a *specific* peer. This helps them embed memory or persistence into the system, enabling handling of state-dependent queries and efficient partitioning of incoming load by having specific nodes handle specific queries. File-sharing systems, object-store systems, and load-balancing among servers in a service-oriented architecture (e.g., Amazon's Dynamo [21]) are example systems that benefit from this functionality. A significant amount of previous research has been performed to solve this problem – e.g., Consistent Hashing [49] and Plaxton et al's approach [76], and the many "distributed hash table" (DHT) approaches inspired by them.

1.2 Bibliography

All of the work presented in this thesis has either been published at peer-reviewed conferences or is currently under submission. Material in Chapter 3 is based on a paper that appeared at Infocom 2006 [106] and partially on a paper that appeared at IPTPS 2008 [98], while material in Chapter 4 is based on a paper that appeared at Usenix 2007 [107]. A paper containing material in Chapter 5 is currently under submission.

CHAPTER 2

RELATED WORK

2.1 Load-balancing in Unstructured P2P Networks

There has been a significant amount of previous work that looks at constructing unstructured graphs, as there has been work work that looks at the problem of randomly selecting peers. Since the problem of random peer selection is intricately tied to the degree distribution in the underlying graph, a few systems propose mechanisms to both construct the graph and to do random selection over the graph – indeed this is our own approach as well (Chapter 3).

2.1.1 Unstructured Graph Construction

Early versions of Gnutella employed very simple graph construction – new nodes contacted Gnutella nodes already in the network, who then flooded their neighbor-discovery requests upto a few hops into the Gnutella network [85, 53]. This join method clearly results in nodes with already high degrees accumulating even more links, thus severely skewing the degree distribution. Indeed, it has been shown that early Gnutella networks followed the power-law degree distribution [85, 1].

Many approaches have previously been proposed for *homogeneous* unstructured graph construction – here all nodes are considered equal, so the approaches ideally give all nodes the same degree. Clearly, these approaches are insufficient in building a heterogeneous graph. Examples are Scamp [31, 32],

Araneola [66], and the Hamilton-cycle based approach by Law and Siu in [57].

Scamp is a gossip approach designed to prevent nodes from requiring full membership knowledge [31, 32]. Each node, on average, learns only about a logarithmic number of other nodes. Scamp does this by having nodes accept neighbor-requests probabilistically, as a function of their current view-sizes, and periodically rebalancing view-sizes. This approach however cannot build graphs where different nodes desire different degrees.

Araneola builds overlays to be used by application-level multicast systems [66]. It aims to build almost-regular graphs where node-degrees are all within the same tight range, and ideally equal to the same value. Each node uses the membership protocol from *lpbcast* [24] to maintain a uniform partial view, and joining nodes use these views to quickly discover neighbors. Like the previous approach, Araneola cannot build heterogeneous graphs as well. An addition concern with Araneola, even in the homogeneous graph construction case, is that it assumes that each newly entering node has access to an independently uniform view of the network – this assumption can be violated under high network churn rates, unless the background gossip required to maintain the uniform views is performed at excessively high rates.

Law and Siu propose a method to build random regular graphs [57] – random regular graphs are likely to be expander graphs, and are attractive candidates for unstructured graphs. This approach maintains a constant number of Hamilton cycles spanning all the nodes in the graph – each node remembers the next and previous node in each cycle. When a new node enters the network or an existing node decides to leave, the Hamilton cycles are quickly reformed to insert the new node or delete the leaving node. Again, this approach cannot

handle heterogeneity, and is also suspect under churn, since it assumes graceful departures by all leaving nodes. Our Swaplinks approach builds on the spirit of this approach to handle churn and accommodate heterogeneity (Chapter 3).

Gia [14] is an unstructured file sharing system that uses random walks rather than flooding to make file searching more efficient and scalable. In contrast to the above schemes, Gia [14] recognizes that different peers have different capacities and accordingly should handle different loads. It does this by making high capacity nodes more "responsible" than low capacity nodes, both by giving high capacity nodes higher degrees and more information to store, and by routing more search queries to them. Gia however does not give fine-grained control over degree or load, something that we would like to achieve.

The Kazaa P2P file-sharing network incorporates the concept of "supernodes" [52]: Each peer here is either an supernode or an ordinary node. Supernodes form the core of the Kazaa network, while ordinary nodes only connect to the Kazaa network through supernodes. Supernodes here generally have good bandwidths and have stable uptimes. Measurement studies have shown that supernodes have orders of magnitude more degree (150-200) than ordinary peers [61]. More recent versions of Gnutella have a similar split between "ultrapeers" and "leaf-peers" [53].

In both of the above cases (Gnutella with ultrapeers and Kazaa with supernodes), the only distinction between different nodes is whether they are supernodes (or ultrapeers) or ordinary peers, so the control over degree and load in this design is coarse – there might be peers whose ideal loads fall between those faced by an supernode and that of an ordinary peer. We would like to achieve more fine-grained control in our designs. Additionally, mechanisms like the

ones we propose in Chapter 3 will be needed in order to achieve good load-balancing within the supernode network itself.

2.1.2 Random Peer Selection

Many gossip-based protocols need to repeatedly pick a node uniformly randomly from the entire population (e.g., [22, 78, 8, 101]). To realize this abstraction, members in the system may need to be aware of all other members currently in the system, as in [78, 101], and as observed in, for example, [8, 47]. Narada [45] is an end-system multicast system that also has all nodes keep track of all other nodes currently in the system, so as to be able to pick random nodes from time to time. While this method of tracking membership may be acceptable in small systems (of upto a few hundred nodes), it will not scale to larger sizes, so is not desirable.

More recent gossip protocols like *lpbcast*[24] and those in [47] use scalable membership schemes to provide the random peer selection functionality. Nodes here repeatedly exchange their uniform "partial views" of the network with other nodes, converging to the case where all nodes have uniform partial views of the network in the ideal case. But these schemes are not designed to handle cases where the desired selection distribution is non-uniform, as would be the case with heterogeneity.

The basic search mechanism used in unstructured P2P systems like Gnutella is essentially a form of repeated random peer selection. Early versions of Gnutella clients flooded search queries a few hops deep into the network, where the contacted peers responded if they had the content being searched for. The

flood mechanism has an inherent shortcoming in large networks [64]: the network and CPU loads grow exponentially with the number of hops, so any value picked for the number of hops is very likely to be either too small (not searching enough hosts) or too large (placing an undue amount of load on the network).

Many researchers have proposed modifications to scale the search mechanism to large network sizes. Almost all of these involve the use of random walks in place of flooding in order to perform file-search. However, just replacing flooding with unbiased random walks, as in the proposal by Lv et al [64], has a drawback: Neither the selection probability nor the query load distribution are uniform, since nodes that have high degrees in the neighborhood graph end up being selected proportionately more often. In addition, the setup here ignores heterogeneity constraints, since the degree distribution does not take into account heterogeneity.

Adamic et al [1] observe that Gnutella-like graph construction methods result in power-law graphs, and propose methods that exploit the heavy skew in degree distribution in such graphs. They propose two differences in the way search queries are handled: (i) Nodes keep track of their neighbors' file-contents as well, and (ii) Search queries are preferentially forwarded to neighbors with high degrees, but in such a way that the query never revisits nodes. This method reduces the number of hops it takes to find a given file, but the resulting skew in the selection and load distribution is even worse than that in the underlying degree distribution. Again, since the underlying graph is not constructed here with heterogeneity in mind, this is not desirable.

Other researchers have proposed schemes to do uniform sampling: Gkantsidis et al [33] build a random regular graph based on the Hamilton-cycle ap-

proach in [57], and run unbiased random walks over the graph. Stutzbach et al [100] use the Metropolis-Hastings method [68, 41] to achieve uniform sampling. Neither of these approaches handle heterogeneity.

Liang and Nahrstedt [62] give a method to do QoS-sensitive peer selection. Peers are grouped into clusters based on their QoS characteristics (like bandwidth, for instance). QoS-sensitive selection is performed here by first picking the appropriate cluster and then picking a random node from that cluster. This approach however requires that the selecting peer know how many nodes there are at each QoS level (e.g., with a particular bandwidth value) to do truly heterogeneity-sensitive selection – in practice this is a hard assumption to satisfy.

Bullet multicast [56] uses a random selection mechanism called RanSub [55]. RanSub operates in waves of network-wide coordinated phases, where in each phase lists of random nodes are distributed through the network. Here, the nodes learned by a given node at a given time are not independent of the nodes learned by another node at the same time. The phases must be run multiple times if different nodes are to ultimately select mutually independent sets of other nodes. In addition, RanSub, as is specified, cannot handle heterogneneous selection.

Zhong et al perform load-balancing by using the Metropolis-Hastings algorithm to preferentially sample nodes with high load and off-loading some of that load to other nodes [113]. This method only works to reduce the imbalance once the imbalance has already set in – we would like to ensure proactive load-balancing. This method also needs nodes to be aware of all of their neighbors' loads.

Two random walk approaches that we closely study in this thesis are Self-Loops [6] and Iterative Scaling [18]. These methods are not suitable to use, as is, for graph construction or to accommodate heterogeneity. In Chapter 3, we discuss how we extend these techniques to adapt them to our setting.

The Metropolis-Hastings algorithm [68] mentioned earlier can be used to achieve desired probabilities of selection over any underlying graph. However this algorithm has a few shortcomings, which we detail in Chapter 3 (Section 3.4).

2.2 Heterogeneous Peer Selection Using Structured Approaches

All structured P2P systems modeled as DHTs (e.g. [80, 99, 86], etc.) assign identifiers to nodes, typically uniformly at random. Random selection in DHTs can be done by randomly choosing a value from the DHT number space, and routing to that value. The problem of random node selection in DHTs, then, can be reduced to the problem of assigning identifiers appropriately.

Even where uniform random selection is desired, assigning a single random identifier to each node is inadequate, because any non-uniformities in the random assignments persist over time. Consistent hashing schemes deal with this by assigning multiple random identifiers[49], and DHTs have proposed something similar, namely creating multiple virtual replicas of each node in the DHT [99].

To achieve heterogeneity, CFS extends this concept, replicating each node a

number of times proportional to its capacity ([20]). This leads to high-capacity nodes having a proportionally larger portion of the identifier space, therefore being proportionally more likely of being selected. The approach of multiple virtual servers per node however entails a blowup in network and computational overheads, and so is not an attractive approach.

There are a number of other approaches that do heterogeneous load-balancing while using the same number of virtual servers per node [79, 34, 114]. These perform load-balancing by transfering responsibility of virtual servers from heavily load nodes to lightly loaded nodes. These schemes (all except the "one-to-one" scheme in [79]) use what could be called rendezvous nodes to help match overloaded and underloaded nodes for the virtual-server transfers. Rao et al [79] and Godfrey et al [34] use a handful of "directory" nodes for this purpose, while Zhu and Hu [114] organize the virtual servers into a tree and use interior nodes within the tree as the rendezvous nodes. The state involved in matching overloaded and underloaded nodes is a concern in these schemes. The few directory nodes need to support all of the load in the first two approaches, whereas the tree in the third approach is vulnerable to churn. An additional concern is the extra load due to the virtual servers, as in the previous paragraph.

A modified multiple virtual node approach is used in Y_0 [35]. Here, virtual node identifiers for each node are selected from a small range of identifiers; the authors utilize the proximity of the node's identifiers to avoid having to maintain separate routing entries for each virtual node. While this scheme is interesting, and a potential candidate for comparison, it has not been analyzed or tested for robustness to high churn. Y_0 also needs all nodes to know (at least

roughly) the number of nodes in the system, which might be an issue under high churn.

Karger and Ruhl propose two schemes in their papers for load balancing in DHTs [51, 50]. The first results in a constant factor bound on ID spaces between successive nodes, but cannot handle the case where the ID spaces are to be split according to capacities. The second scheme looks at item load balancing, where the number of items that are stored at any node should be within bounds and dependent on node capacity. Nodes periodically randomly probe other nodes, and share load by transfering items from heavily loaded nodes to lightly loaded nodes. With minor variations, we could modify this scheme to split ID space according to node capacities and run over the Bamboo DHT – we describe this *KRB* scheme in Chapter 4. Rao et al's "one-to-one" scheme [79] is very similar to the item-balancing approach, but is dependent on multiple virtual servers per node, and does not handle nodes leaving the network. Shen and Xu's approach [94] also is similar in spirit to the item-balancing approach, with a few extensions. We use KRB as the candidate structured approach for our performance comparisons.

Ledlie and Seltzer [59] present the *k-choices* algorithm for load balancing in settings with skewed query distributions and heterogeneous capacities. *k-choices* is similar to KRB, in that both place nodes at IDs that minimize load imbalance. The difference is that *k-choices* assumes that each node knows its absolute desired load, whereas in KRB, nodes only have a notion of relative desired load.

Other researchers have proposed schemes to evenly split the load among all nodes. In Byers et al's item-balancing approach, before an item is placed in

the DHT, multiple randomly picked peers are probed for their loads, and the item is placed at the least loaded location [10]. In Bienkowski et al's ID-space balancing approach [7], nodes probe one another, and decide to leave and rejoin the DHT at new spots if doing so would help reduce the ID-space imbalance. This is similar to our adaptation of Karger-Ruhl's approach in KRB. Neither of these schemes however handle heterogeneity.

Accordion [60] and HeteroPastry [12] give schemes that tailor nodes' degrees and their message loads according to capacity and network activity. These schemes however do not provide capacity dependent namespace partitioning, and so cannot support heterogeneous random selection by routing to uniformly randomly selected IDs. An alternative approach might have been to use unbiased random walks over these networks for random selection, but the control over degrees in these schemes is not fine-grained enough (i.e., average node degrees are not proportional to capacities) for this to result in the desired selection distribution[1].

2.3 Finding the nearest peer

In Chapter 5, we tackle the problem of finding the nearest peer in P2P networks, specifically in the case where a peer's nearest peer is in the same extended LAN or campus network. In that chapter, we also identify the *clustering condition*, where a large number of end-host networks are all at about the same latency from one another. We show in Chapter 5 that this condition presents an obstacle in finding the nearest peer, and present mechanisms to help overcome it. We

[1]To be fair, neither of these schemes expressly aim to maintain node degrees perfectly proportional to capacities.

now discuss previously proposed schemes aimed at finding the nearest peer, and how they fare under the clustering condition.

Karger-Ruhl's scheme to find the nearest peer [48] and Meridian's closest node algorithm [111] are what could be called *Distance-based sampling schemes*. Each peer here picks neighbors based on its distances to them: the concentration of neighbors is high at small latencies, and drops off at larger latencies. When peer P is handling a request to find the nearest peer to another peer N, P would forward the request to a set of neighbors at distances that are a function of the distance between P and N. Intuitively, if the request is forwarded to those neighbors that are at about the same distance as P is to N, then by random chance, one of them is likely to be closer to N than P is. However, assumptions made by these schemes about the distribution of latencies – the *doubling* assumption made by Meridian, and the *growth-restricted* assumption by Karger and Ruhl – are violated under the clustering condition. Please see Section 5.2.3 in Chapter 5 for a more detailed discussion of why Meridian is unable to find the nearest peer under the clustering condition. An almost identical explanation holds for Karger-Ruhl's scheme.

In Tapestry [42], peers arrange neighbors in different "levels", with exponentially fewer choices for neighbors as the level increases. The idea is to pick, in each level, a fixed number of neighbors that are closest to the peer from among the available choices for the level. The levels are built up iteratively, starting from the highest level, i.e., the level with the fewest eligible neighbors that can occupy the level. The level i neighbors of peer P are chosen from among appropriate neighbors of its level $i+1$ neighbors. The peers are assumed to be in a metric space where the above iterative construction succeeds in finding the

closest eligible neighbors at each level. The closest neighbor overall is the closest neighbor in the lowest level. Castro et al propose a similar method to find the nearest neighbor in the Pastry overlay [13]. Both of these are practical realizations of Plaxton et al's original proposal in [76], where the required invariants of neighbors at each level were stipulated, but no methods were given to maintain the invariants in a dynamic setting.

Now consider Tapestry under the clustering condition, where say a new peer N is in the same end-host network as another peer P already in the system. The new peer's search for its nearest peer might reach one of the peers inside its cluster, i.e., one of the levels might include neighbors in its cluster [2]. But it is unlikely that it will then proceed to discover P. This is because all the peers in the cluster except P look almost identical to N, so the only way N would discover P is by first picking as its neighbor a peer that has P as a neighbor in the appropriate level; the likelihood of this latter event happening is small.

Practical Internet Coordinates [17], Fonseca et al [25], Zhu et al [114], and Shen et al [94] all propose schemes that use *network coordinates* to estimate latencies between any two arbitrary peers, and leverage the coordinates to discover the closest peer. Once each node computes its own coordinates based on a few measurements, it can estimate whether it is close to any other node with valid coordinates without having to initate new pair-wise measurements. There have been many other network coordinate schemes that have been proposed, e.g., Global Network Positioning [70], Big-Bang Simulation [93], Lighthouse [74], Network Positioning System [71], Vivaldi [19], Mithos [108], Virtual Landmarks [102], Internet Coordinate System [63], PCoord [109], etc., all

[2] A cluster here is the set of peers in end-host networks all at about the same latency from one another (Section 5.2.1).

of which can potentially be used to find the closest peer as well. Distributed binning [82] is similar in vein, but instead of coordinates, uses *bin numbers* that indicate peers' relative latencies to a given set of landmarks. An assumption with these schemes is that the population of peers is embeddable into a space with a small enough number of dimensions that the coordinate scheme is accurate and practical, and the coordinates can be reliably used to find the closest peer. However, this is not the case under the clustering condition: The number of dimensions needed to embed all the peers in the cluster is very large, on the order of the number of the number of networks in the cluster.

Internet Distance Maps (IDMaps) [27] is another approach to estimate distances between end-hosts without direct measurements between them. IDMaps deploys a number of *tracers* at various locations and has them track latencies between one another. End-hosts in turn measure their own latencies to nearby tracers. Latency between two end-hosts is now estimated as the sum of the latencies between the end-hosts and their respective tracers and the latency between the tracers. Other related approaches include Dynamic Distance Maps [103], Shavitt et al's approach [92], and Internet Iso-bar [15]. As proposed however, these schemes do not offer a mechanism to find the nearest peer; one possible approach to do it is to group end-hosts that are close to the same tracer, and search among this group for nearest peers. But under the clustering condition, there would be a large number of peers that are all equally close to a given tracer (unless tracers were installed in all end-networks, an infeasible proposition), so it would be expensive to find the exact-closest peer.

Tiers [5] is a hierarchical scheme to find the nearest peer. The Tiers hierarchy consists of multiple levels: The lowest level has all the peers in the system,

with nearby peers grouped into clusters. A single peer from each cluster is chosen as the cluster's representative. Each cluster representative is part of the next (higher) level, where again the member hosts are grouped into clusters and representatives chosen for these clusters. This continues until the topmost level, which has just a single cluster. When a new peer joins the system, its search for the nearest peer starts from the topmost cluster. The new peer measures its latencies to each of the nodes in the cluster, and picks the one that it is closest to, and the search continues with the cluster (in the next lower level) represented by the picked peer. The search eventually reaches a cluster in the lowest level, and the nearest peer in the cluster is chosen as the nearest peer overall.

Before we discuss the behavior of Tiers under the clustering condition, we need to distinguish a cluster formed by Tiers from the cluster of peers that forms the basis of the clustering condition. We refer to the former as a *Tiers-cluster*, and to the latter as a *peer-cluster*. Tiers forms multiple Tiers-clusters at the lowest level from each peer-cluster. This means that multiple peers from the same cluster also occupy higher levels in Tiers. When a new peer traverses down the hierarchy looking for its nearest peer, it will eventually select its nearest peer in the same end-network only if it picks the right cluster-representative at each step of the hierarchy. Since this essentially reduces to random choices at each step, it is unlikely to succeed in finding the exact-closest peer in the same end-network.

In contrast to the above approaches, centralized approaches using *beacon-servers* are suggested by Guyton et al [40] to find the nearest replicated server to a client and by Beaconing [54] to find the closest peer. In the former, each of the beacon servers measure their latencies to each of the servers, and the client

that wishes to find its nearest server. They estimate the latency between the client and each of the servers using Hotz's metric [43] based on triangulation bounds. The server with the least estimated latency is returned as the closest server. When this approach is used to find the closest *peer*, the beacon-servers now track latencies to all peers. But under the clustering condition, this leads to most peers in the same cluster but different end-networks having almost identical latencies to all the beacon servers, since most end-networks would not have a beacon server deployed in them. It follows that all such peers are impossible to tell apart.

In Beaconing, each beacon server tracks and remembers its latency to each peer. When a new peer P wants to find its closest peer, each beacon returns the set of other peers that are at about the latency to itself as P is. P then probes the peers in the returned sets, and picks the closest among these. Sharma et al propose a similar approach in Netvigator to find the nearest peer [91]. Again, under the clustering condition, this leads to the beacon servers having the same latencies to most peers in a cluster, making them indistinguishable from one another.

CHAPTER 3

HETEROGENEOUS OVERLAY CONSTRUCTION AND RANDOM NODE SELECTION IN UNSTRUCTURED P2P NETWORKS

3.1 Introduction

Unstructured P2P networks do not impose rules on which nodes can be neighbors in the underlying graph, beyond restrictions on the degrees of individual nodes. Thus we can model the underlying graph simply as a random graph.[1] Unstructured P2P applications have dominated usage in the history of P2P systems. Examples of unstructured applications include the Gnutella, Kazaa, and BitTorrent file-sharing systems, P2P multicast systems like Chunkyspread [104] and Chainsaw [72], and others.

In this chapter, we address the broad problem of load-balancing in unstructured P2P applications. We specifically examine load-balancing in *heterogeneous* settings, where the capacity to bear application-imposed load varies between the different nodes. Accordingly, our aim here is to ensure that the load imposed by the application on each node is proportional to the capacity of the node to support the load. The load imposed could be along any of the different dimensions of bandwidth, CPU, disk, etc. Since most P2P applications predominantly impose load along just *one* of the aforementioned dimensions (e.g., bandwidth, CPU, disk, etc.), we assume in this work that the load imposed at any given node can be captured by a single unidimensional value.

[1] By random graph, we mean a graph where the end-points of each edge are randomly selected from the entire population, with restrictions on the number of edges adjacent to each node.

In this chapter, we examine two important "drivers" of load seen by nodes in a variety of unstructured P2P applications: (i) the node-degree in the underlying neighborhood graph, and (ii) the process of *random selection*, where each time there is application load to be handled, a node from the population of peers is randomly selected to handle the load. We develop and adapt solutions that use random walks to ensure that the resultant load due to both of these factors is in accordance with nodes' capacities to handle them [2]. Note that we do not assume perfect control over resultant load: there is a significant random component in our algorithms and graphs, since they use random walks. Rather, we expect statistical control, along the lines of what would be possible with true random selection. Below we detail each of the two load-causing factors in turn.

Degree in underlying graph: In many unstructured multicast systems (e.g., Chunkyspread [104], Chainsaw [72]), file-sharing systems (e.g., Gnutella, Gia [14]), and gossip-based protocols (e.g., lpbcast [24], Scamp [32]), the higher the degree of a node is in the underlying random graph, the greater is the application load seen by the node. There is a certain cost to maintaining a link in the underlying graph as well – for instance in the periodic keep-alive messages used to determine if a neighbor node is still active – which also increases the overall resultant load with node-degree.

Random peer selection: Nodes in unstructured systems typically do not keep any state about the rest of the system other than who their neighbors are. Thus when there is load to be handled (e.g., a file-name to be searched for), such applications essentially randomly select peers from the population to handle the load. We want control over the load that results from this process of

[2]While we study only unstructured algorithms in this chapter, we will be comparing the best unstructured approach, called Swaplinks, with a DHT-based approach in the next chapter.

random peer selection. Given that our solutions use random walks to perform the random selection, we want control over the probability that a node will be *visited* in random walks, and the probability that a node will be *selected* in random walks. A node is selected when a random walk ends at that node. A node is visited when a random walk traverses that node during a walk.

Control over visits is important for two reasons. First, nodes experience load every time a node visits them. If we wish to have control over load, then we correspondingly need to have control over how often nodes are visited. Second, some applications execute application functionality every time a node is visited during a walk. For instance, in many file search algorithms (e.g.,[1, 64, 14, 16]), each node visited is searched for the desired key words.

A number of applications require random node selection as a way of configuring application-specific topologies. This selection should follow a certain desired probability distribution. Examples of these include overlay multicast or file distribution applications [26, 56, 9], file sharing applications [64, 14], and "proximity addressing" applications [19, 17]. (The latter is an application where nodes form addresses that can be used to indicate how close nodes are to each other in the network.) As we show later in this chapter, many of the above effects can be obtained by establishing the appropriate node degree in the underlying random graph, and by using random walks with appropriate control over the nodes selected.

Note that some applications (e.g., overlay multicast applications like Yoid [26]) use two separate graphs: one for normal operation (e.g., the multicast graph), and the other a random graph of the kind we discuss in this chapter. The first graph is built keeping constraints like network proximity in mind, and

thus is not completely random, and therefore not as robust as a random graph. Maintaining the second (random) graph makes the application more robust to partition, while also giving the application the ability to select random nodes in the graph.

Given the variety of unstructured applications that require heterogeneous random graph construction and random peer selection, we develop a single set of algorithms that can provide both of these functionalities. In addition to having good control over degree, load, etc., we require our algorithms to also be scalable and robust to churn: this is a natural requirement for any P2P system. And finally, a key requirement that permeates all of our work is that of simplicity. This requirement goes beyond the basic notion that, all other things being equal, simple is better than complex. We believe that algorithmic simplicity is central to achieving scalability. Our intuition is that, as networks grow, more complex algorithms will exhibit more failure modes and ultimately limit scalability even where the basic algorithms scale according to traditional measures such as memory and message overhead.

Our full set of requirements for a random graph building and node selection mechanism are therefore as follows: scalability, simplicity, robustness, control over node selection probability, control over node degree, and control over message load. A mechanism satisfying these properties can then serve as a foundation for numerous unstructured P2P applications.

We conclude this section by listing the two broad contributions we make in this chapter: (i) We design heterogeneous graph building and random node selection algorithms that are practical to deploy and that are functional over a wide range of requirements. Towards that end, we explore a number of funda-

mental approaches, including variations on existing approaches as well as new approaches. (ii) Using simulations, we provide a broad comparison of the various approaches. In so doing, we identify our novel technique, called Swaplinks, as the most attractive graph construction mechanism from a practical point of view.

The rest of the chapter is organized as follows. Section 3.2 describes how joining nodes get to know of already existing nodes in the graph. Sections 3.3 and 3.4 describe the four basic approaches for both building graphs and walking them. All of these approaches need nodes to know their target degrees; Section 3.5 describes how nodes can compute their target degrees in a distributed fashion. Section 3.6 presents detailed results of simulations used to evaluate the performance of the four approaches. Section 3.7 concludes.

3.2 Initial Node Discovery

Any new node that wants to join the graph needs to know at least one already existing member in the graph. While our algorithms work with any scheme that helps new nodes discover existing nodes, a practical and simple approach we envision for this purpose is to establish a *rendezvous* node at a well known location (a DNS name or IP address). Joining nodes first contact the rendezvous node, which tells them of previously joined nodes. The rendezvous node could tell joining nodes about the same small set of joined nodes, but this puts an undue load on those joined nodes. The rendezvous node could remember all joining nodes, and tell the joining node of some small random subset, but this puts an undue burden on the rendezvous node.

Therefore, we assume a very light-weight approach whereby the rendezvous node remembers a small set (of about 10) of the most recently joined nodes. New nodes enter the network by contacting some (or all) of these nodes. This approach effectively spreads the load of node discovery. Note, however, that even this approach requires caution, because with a naïve graph-building scheme, this approach can lead to "long-thin" (high diameter) networks. Nevertheless, in the remainder of this chapter, we assume this style of node discovery. This discovery mechanism can be made more robust by having the rendezvous node remember an additional small set of stable random nodes known to be up with high probability: the rendezvous node discovers such nodes using selection walks. The results presented in this chapter however do not assume the more robust scheme.

Note that the rendezvous node can be replicated, and one can be selected by the joining node using DNS or even IP anycast [73]. In this case, however, the rendezvous nodes must take care to keep each other informed of the initial joining nodes so as to avoid a graph partition in the early stages of its formation.

3.3 Algorithms for graph construction

In this section, we describe the basic random walk approaches we use in building graphs. We start off the discussion by first considering the homogeneous case, where all nodes wish to have the same degree and probability of selection. A truly unbiased walk, whereby each node selects uniformly randomly among its neighbors, will select high degree nodes proportionally more often than low degree nodes simply because more links lead to those high degree

nodes. Therefore, unless the graph has perfectly uniform node degrees, the random walk must somehow be biased against high degree nodes.

While this is true both for walks used for the purpose of selecting nodes to build the graph (*build walks*), and for walks used for other node selection (*selection walks*), the problem is more severe for build walks. This is because any favoring of high-degree nodes by build walks will compound itself as the network grows. If a node obtains a slightly higher than average node degree, the subsequent joining nodes will select it more often and choose it as their neighbor, thus giving it an even higher node degree, thus making it a target for yet more neighbors. Indeed, it is not enough for build walks to simply negate the effect of node degree, so that selection is uniform. The reason for this is that early joining nodes participate in more "selection trials"– they get more chances to be selected as neighbors by joining nodes than do later nodes. Therefore, there must be additional bias or mechanisms to limit the rate at which high degree nodes collect more links.

Now, when we include the case of heterogeneous node capacities, we require random graphs where higher capacity nodes have proportionally higher node degrees than lower capacity nodes. Further we require that walks visit and select nodes in proportion to their capacities.

Our basic approach to building graphs is for each node i in the graph to establish a fixed number of links K_i, called *outlinks*, with randomly selected nodes in the graph[3]. To achieve heterogeneity, high-capacity nodes establish more outlinks than low-capacity nodes. For instance, if the lowest capacity node establishes 5 outlinks, a node with twice that capacity will establish 10 outlinks. Ex-

[3]We make only a logical distinction between outlinks and inlinks, for control over degree distribution. In particular, message flow can occur in either direction over any link.

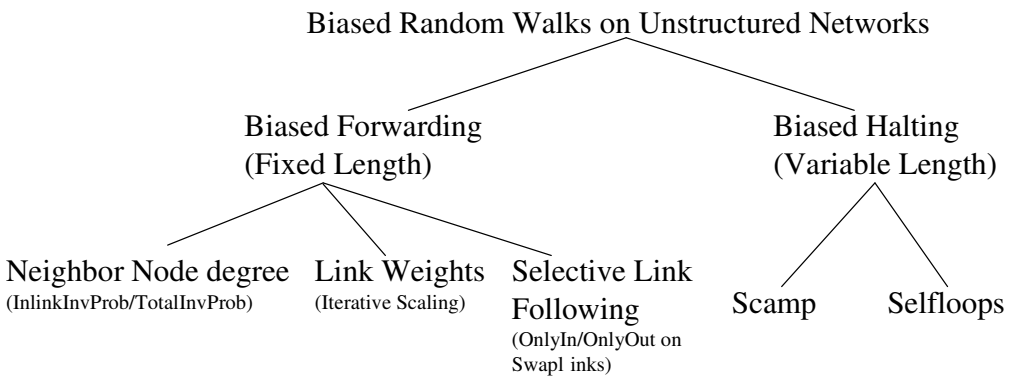


Figure 3.1: Classification of Biased Random Walks.

actly how many outlinks a node should establish is a function of both the node's own capacity and the distribution of capacities among the rest of the node population. We give a brief discussion of how peers can compute this figure in Section 3.5. The rest of this chapter will assume that nodes have computed their ideal outdegrees, and formed the required number of outlinks. The goal of each of our graph construction mechanisms is therefore for nodes to obtain roughly as many inlinks as they have outlinks (within random variations). We refer to this as the *expected node degree* or *expected indegree*.

There are two fundamental approaches to counteract the effects of early joiners obtaining more inlinks, and the self-reinforcing trend of high-indegree nodes becoming even higher-indegree nodes[4]. One approach is to simply endow build walks with an even stronger bias against high-indegree nodes, so that most nodes never get high indegrees. There are several ways to do this, which are shown in the taxonomy of Figure 3.1. The other approach is to actively manage each node's indegrees, so that nodes explicitly shed inlinks when they get too many. The basic mechanism, which we call *Swaplinks*, is for nodes with high indegrees to move an inlink to nodes with low indegrees. We next describe the

[4]In general, when we say "high indegree", we mean "higher-than-expected indegree".

taxonomy of the biased walk approaches, and then go on to describe each of the biased walk approaches we study in this chapter. Finally we discuss the Swaplinks approach in Section 3.3.5.

Taxonomy of Biased walks:
In our biased walk approaches, the basic graph building mechanism is for a joining node i to establish and maintain a constant number of outlinks K_i with nodes discovered by taking K_i random build walks. If an outlink is lost, for instance because the neighbor crashes or leaves the network, the node reestablishes the outlink by taking another biased random walk and adding an outlink to the discovered neighbor.

Note that in all our biased walk approaches, a node never has the option of refusing a request to create an inlink. One could easily imagine a scheme where we could do this, for instance by not terminating a build walk at a node if its indegree-to-outdegree ratio is above some constant. We chose not to consider such approaches in part because the bias tends to prevent the need for this, and in part because we wanted to keep our approaches simple so that we could better understand their fundamental characteristics.

Note also that any given random walk may fail, for instance because of packet loss or sudden node failure. In this chapter, we assume that any node initiating a walk will repeat it if it does not succeed within some short time.

Looking at the taxonomy in Figure 3.1, we see that there are two fundamental ways to bias a walk, which we call biased-halting and biased-forwarding. In biased-halting, the next hop at a node is picked uniformly at random from among all of the links at the node – there is no weighting in this regard. Instead,

the walk is ended at each node with a random probability that is weighted inversely to the degree of the node. The result is that the length of each walk is variable, though the average length can be fixed. We discuss the *SelfLoops* style of biased-halting walk in section 3.3.1. Scamp [31, 32], briefly discussed in Section 2.1, also uses biased halting walks to find neighbors for newly entering nodes; we do not discuss Scamp further here, but we do briefly evaluate it later.

In biased-forwarding, the selection of the next hop in the walk is weighted against high degree nodes. In these walks, the number of hops is set at a fixed constant H, which must be long enough to allow the walk to *mix* into the network – a constant times the diameter of the network[5]. The biased-forwarding walks we study are *InlinkInvProb*, *TotalInvProb*, and *Iterative Scaling*, discussed in Sections 3.3.2 – 3.3.3.

There are trade-offs between the biased-halting and biased-forwarding approaches. On the one hand, biased-forwarding requires nodes to exchange state about their neighbors–their node degree or a more general weighting. Biased-halting requires no special knowledge of neighbors. On the other hand, biased-halting walks tend to unfairly load high-degree nodes, because a large number of walks tend to be forwarded to high-degree nodes, only to continue on with high probability.

[5]Given that the diameter grows slowly with the size of the graph, and given the range of network sizes and node degrees this thesis examines, we can simply pick a conservative value like $H = 10$.

3.3.1 SelfLoops (SL)

Biased-halting approaches are ideal in settings where the graph is not under one's control, and the cost of calculating weightings is high. Indeed, the biased-halting approach we use is based on work by Bar-Yossef et al [6], who used it to select web pages with uniform probability. Their approach, which we call *Self-Loops*, is elegant and intuitively appealing. The basic idea is to emulate a graph with perfectly uniform node degrees by adding virtual links to oneself (i.e. self loops!) For example, say that the target uniform node degree (the virtual degree) is 100. A node with 90 real links would add 10 virtual links to itself. A node with 25 real links would add 75 virtual links to itself. Subsequently during a walk, each "link" is selected with equal probability, and the virtual walks are of "fixed length", though the real walks are not. In practice, for uniform selection, the virtual degree is set to a large constant at each node, and the value used for the virtual hop length can be set such that the average real hop length is as needed.

The Bar-Yossef approach, as defined, does not support heterogeneity or provide the needed bias for build walks. We modify the Bar-Yossef approach as follows to make it useful in our setting. For selection walks, the virtual degree of each node is made directly proportional to its outdegree. For build walks, the virtual degree is directly proportional to the square of the outdegree and inversely proportional to the indegree ($\frac{od^2}{id}$) (see Table 3.1). To see how this leads to the desired degree distribution, we first need to introduce the idea of *refreshes*: A refresh is where a node discards one of its outlinks and chooses another (also see Section 3.3.4). Assuming a graph where all nodes have performed a large number of these refreshes, and have reached a stable degree distribution, let us

examine the change in indegree of node i when a node chosen uniformly at random from the entire graph performs a refresh. Since the steady state has been reached, the net change in the expected indegree of i due to the refresh is zero. Node i loses an inlink if the discarded link happened to be its inlink, so the probability that i loses an inlink because of the refresh is given by $c \cdot indeg(i)$, where c is a constant. The probability that i gains an inlink because of the refresh is given by $c' \frac{outdeg(i)^2}{indeg(i)}$. These two probabilities need to be equal, so we get $indeg(i) = c'' \cdot outdeg(i)$, where the constant of proportionality is of course 1. We show later through simulations that the linear dependence of indegree on outdegree is achieved even without refreshes.

With this modification though, it gets much harder to estimate the virtual hop length to use to achieve a desired average real hop-length during graph construction. A conservative option is to use a large enough value, but this results in a larger average hop-length. In our experiments, we use trial and error to estimate the virtual hop-length. This lack of tight control on the average hop-length is a drawback of the SelfLoops approach.

Note that one of the problems with biased-halting walks is that any given walk can be quite short. For instance, if the walk length is set to terminate after an average of 10 hops, then there is a very small chance that a walk will end at one hop, a bigger chance the walk will end within two hops, and so on. Such short walks clearly do not mix well, so we experimented with a hybrid approach where if the expected walk length was h hops, the walk could not terminate within $h/2$ hops. For the first $h/2$ hops, we use one of the biased-forwarding walks described below (specifically the TotalInvProb walk), and for the later half we use SelfLoops. We call this hybrid TotalInvProb-SelfLoops (Hyb-TIP-

Table 3.1: Summary of the different walk strategies for a walk at node A; node N is a neighbor of A, and w_N^A is the probability that a walk at A is forwarded to N. $virt - deg$ denotes the virtual degree.

Name	Link Weighting
Swaplinks(SW)–Normal/Outlink-loss	$w_N^A = \frac{1}{indeg(A)}$ if $N \in$In-nbrs(A)
Swaplinks(SW)–Inlink-loss	$w_N^A = \frac{1}{outdeg(A)}$ if $N \in$Out-nbrs(A)
InlinkInvProb(IP)	$w_N^A \propto \frac{outdeg(N)}{indeg(N)}$; $\Sigma_{N \in Nbrs(A)} w_N^A = 1$
TotalInvProb(TIP)	$w_N^A \propto \frac{outdeg(N)}{totaldeg(N)}$; $\Sigma_{N \in Nbrs(A)} w_N^A = 1$
SelfLoops(SL)-Selection	$w_N^A \propto \frac{1}{virt-deg(A)}$; $virt - deg(A) \propto outdeg(A)$
SelfLoops(SL)-Build	$w_N^A \propto \frac{1}{virt-deg(A)}$; $virt - deg(A) \propto \frac{outdeg(A)^2}{indeg(A)}$
Iterative Scaling(IS)-Selection	$\Sigma_N w_N^A = 1$; $\Sigma_N (outdeg(N) \cdot w_A^N) = outdeg(A)$
Iterative Scaling(IS)-Build	$\Sigma_N w_N^A = 1$; $\Sigma_N \left(w_A^N \cdot \frac{outdeg(N)^2}{indeg(N)} \right) = \frac{outdeg(A)^2}{indeg(A)}$

SL).

3.3.2 The Inverse-Probability walks

In this style of biased-forwarding walks, the probability of forwarding a walk to a node is directly proportional to the outdegree of the node and inversely proportional to either its indegree (*InlinkInvProb*, or IP) or the total degree (*To-*

talInvProb, or TIP). IP produces a stronger bias, and is used for build walks, while TIP is used for selection walks.

Note that one could invent any number of inverse weightings derived from neighbor node degree (square of the degree, square root of the degree, etc.). Though we did explore these variations, we found the above approaches (IP and TIP) to be adequate for our purposes, and therefore do not report any of the other variations in this chapter.

3.3.3 Iterative Scaling (IS)

In this next style of biased-forwarding walk, an iterative distributed computation is executed across all nodes, allowing the nodes to assign weights to all links. The computation, called *Iterative Scaling* (IS), is based on a technique used to derive the elements of a matrix when the row and column sums are known [18]. Scamp applied this technique to random walks as a means of randomly selecting an "introducer" node that helps a newly entering node join the network [32]. To employ the IS scheme in a graph setting, each node assigns outgoing and incoming weights to each of its links, where the outgoing weight of a link corresponds to the probability that the link is picked during a random walk from the node, and the incoming weight corresponds to the node's perception of the probability that it is picked during a random walk from the other end of the link.

Nodes periodically normalize their weights by scaling their incoming (outgoing) weights so that the incoming (outgoing) weights add to 1, and exchange weights through updates: when node A receives a weight update from neigh-

bor B for the link A-B (denoted l), A would set $wt^A_{in}(l) = wt^B_{out}(l)$ and vice-versa. ($wt^A_{in}(l)$ denotes the incoming weight assigned by A to link l). The weight scalings and updates are intended to bring the system to a state where at every node both the incoming and outgoing weights add to 1 each, so a sufficiently long random walk is equally likely to end at any node.

To accommodate heterogeneity and the different biases for build and select walks, we modify the IS approach similarly to how we modified the Bar-Yossef approach. When used for selection, the ideal probability that a node is selected is proportional to its outdegree. When used for building, the ideal value is directly proportional to the square of the outdegree and inversely proportional to its indegree. So, when weight updates are performed at a node A, the incoming weight for each link A-B is scaled by the estimated probability of a walk reaching B (which is $k \cdot outdeg(B)$ for selections and $k \cdot \frac{outdeg(B)^2}{indeg(B)}$ for graph build) before the normalization is performed.

3.3.4 Some Issues with Biased Walk Approaches

Exchanging neighbor information: Given that the biased-forwarding schemes require nodes to have knowledge about their neighbors–explicit with inverse probability (IP), implicit with iterative scaling (IS)–we must address the question of how this knowledge is obtained. At one extreme, with IS we could run the distributed computation to steady state every time there is a link change somewhere. This is obviously not practical, as links may come and go at a rapid rate, and not really necessary either because in any event the effect of a link change diminishes rapidly with distance from the link. With IP or IS we could

have each node send a message to all of its neighbors every time it experiences a link change. This is still somewhat heavyweight, but certainly reasonable. A third approach is to simply piggyback the neighbor information on other messages. This will result in less accuracy, but is simpler and more efficient.

Note that it may or may not be possible to piggyback all neighbor information on the periodic keep-alive messages used by nodes to determine if neighbors are still up. The reason is that, for high-degree nodes (or for nodes that belong to a large number of low-degree graphs), it is easy to imagine an optimization whereby only a few of a node's many neighbors probe for liveness. These few neighbors would then tell the remaining neighbors if the node went down. [6] In this case, the node obviously cannot convey periodic information to most of its neighbors.

Graph refreshes: As described above, build walks have a stronger bias in order to counteract the effect of early joining nodes having more opportunities to obtain neighbors. One of the effects of this bias is that joining nodes have a higher probability of attaching to more recently joined nodes than old nodes, thus removing some of the randomness from the graph. And, in spite of the bias, older nodes inevitably accumulate more links (as described earlier); this too harms the degree distribution in the graph. One way to counteract this is for nodes to periodically remove an outlink and replace it with another randomly selected node. We call this process *refreshing*. As our results show, refreshing can have a strong improvement on the quality of the graph.

Refreshing has a number of negative aspects though. One is its overhead. Another is that graph changes may negatively affect the application using the

[6] We describe a "smart-pinging" scheme that does precisely this in Section 4.4.4.

graph. A third is simply that it introduces a new engineering requirement into the system. With refreshing, one now has to ask how often to refresh, when it is no longer necessary to refresh, and so on. All things being equal, it is better not to have to ask and answer these questions.

Note that churn, where nodes leave the network, has the same effect as refreshing.

3.3.5 Swaplinks (SW)

Swaplinks is inspired by, but quite different from, the approach used to build random graphs by Law and Siu [57]. The basic idea in [57] is that when a joining node A adds an outlink to a node B discovered during a build walk, one of the inlinks of node B is transfered to node A. This has the effect of maintaining a constant number of inlinks at node B, and of giving the joining node A the same number of inlinks as outlinks, which is our goal. Indeed, if a graph only grows (nodes never leave), then every node will have an indegree that is equal to its outdegree.

The wrinkle to this approach is when nodes leave. If we want to maintain the invariant of all nodes having exactly the expected indegree, as Law and Siu do, then the procedure becomes quite complex. Law and Siu's approach to handling node departures is to have each departing node help all of its neighbors form new links so that the invariant is maintained after the departure as well. This approach fails in the presence of abrupt (non-graceful) node failures. To make this robust against abrupt departures, we might need to have each node know some or all of its neighbors' neighbors, but then this will fail in the

presence of simultaneous multiple departures. Dealing with all of this would require additional mechanisms not specified by [57], and makes this approach unattractive.

However, if we relax the constraint of having to maintain the perfect indegree invariant at all points of time, then the problem of handling churn becomes much more tractable. Before we discuss how our Swaplinks technique handles churn, we need to provide definitions of two kinds of walks used solely with Swaplinks:

OnlyInLinks: This is one type of random walk that is essentially a biased-forwarding walk, but in fact requires no knowledge about the neighbors. In this fixed-length walk, each node chooses uniformly randomly among its inlinks only. The idea here is that when the indegrees of nodes are close to the outdegrees, walking only inlinks results in selection roughly proportional to each node's outdegree. OnlyInLinks itself though cannot be used to build graphs, because the rendezvous server would return a list of the most recently joined nodes, and since all links point from new nodes to older nodes without refreshes, walking only the inlinks would never take the walk outside this set of recent nodes. The end result would be a "long and skinny" network, one with a large diameter, and therefore not desirable.

OnlyOutLinks: This is the analogous walk where each node chooses uniformly randomly among its outlinks. The OnlyOutLinks walk selects nodes with high indegrees with greater probability.

The Swaplinks approach works as follows. When a node joins, it follows the procedure described above – for every node with which it forms an outlink, it

steals one randomly selected inlink. The build walk used for selecting the node is OnlyInLinks. This works in this case because the swapping of links mixes the graph sufficiently to completely avoid any trend towards newly joined nodes.

If a node A loses an outlink (due to node deletion), then it replaces the outlink with a new neighbor O discovered with an OnlyInLinks build walk. Unlike the case of a new node join though, now O does not donate any of its inlinks to A, as A is not looking for inlinks here. Analogously, when a node B loses an inlink due to a node departure, B checks if its indegree is less than its outdegree. If so, it needs to establish a new inlink. It does this by launching an OnlyOutLinks walk to discover node I that is likely to have high indegree. A randomly selected in-neighbor of I now discards its outlink with I, and forms a new outlink with B. [7]

Now consider a sequence of node deletions. Assuming that the indegrees of the deleted nodes are close to the respective outdegrees, we will have roughly the same number of broken outlinks and broken inlinks as a result of the deletions. Now when a node A repairs its broken outlink, it forms a new outlink to a new neighbor O, thus increasing O's indegree, in turn increasing the likelihood that O is chosen by an OnlyOutLinks walk to replace a broken inlink of some other node, which results in O's indegree dropping back to its earlier value. Thus the churn-handling mechanism described above ensures that the degree distribution never gets too far from the desired distribution, even after a long sequence of node departures. (Section 3.6 has the related results.)

Swaplinks has a certain engineering appeal when compared to the biased walk approaches. In particular, there are no engineering decisions required

[7] Note that a walk is initiated here only if some node departure led to a link loss; in the above instance, I will not launch any walks as a result of its losing its inlink to B.

about how to exchange information between nodes (as in biased-forwarding), and how often to refresh (as in both biased-forwarding and biased-halting), and no uncertainty about how long walks may take (as in biased-halting). Perhaps the only negative of Swaplinks is that there is extra overhead when a node leaves, because sometimes two walks must be taken (to replace both outlinks and inlinks) instead of just one.

3.4 Selection Walks

The previous section focused on graph building. The four walks described, however, can be used for selection over any of the graphs – how a graph is walked is independent of how it is built (assuming that the necessary neighbor information is exchanged during building). To summarize, they are Total Inverse Prob (TIP), Iterative Scaling (IS), SelfLoops (SL), and the hybrid TIP-SL(Hyb-TIP-SL).

There is an important limitation to the SL and hybrid TIP-SL approaches that result from the fact that SL is a biased-halting scheme and therefore has variable length walks. Specifically, the file sharing applications described in Section 3.1 require long walks where work is done (a local file search) at each node visited. SL walks, however, do not achieve the desired selection distribution during the walk, as each step is unbiased. Rather, they only exhibit the desired selection distribution upon ending.

While the file sharing application is an important one, more generally the notion of a node starting a walk from the node where the last walk ended, instead of from itself, is useful. We refer to these types of walks as *cursor* walks,

due to the fact that the last node visited can be seen as a cursor pointing to where to start next. The cursor walk works as follows: the node initiating the walk remembers the previously selected node P, and when the next selection is to be performed, takes a short (1 to a few hops) walk from P, instead of starting each walk from itself. The first random selection here is performed in the usual non-cursor manner, and the subsequent selections are performed using the cursor.

In addition to being suitable for applications like the file-sharing application, the cursor approach reduces the imposed load and latency by an order of magnitude, at the cost of maintaining information about the cursor. Further, by spreading the selection load uniformly across the network, it improves the load balance in scenarios where a small set of nodes initiate the majority of the random walks, whereas in the non-cursor approach the initial load during any random walk is necessarily borne by nodes close to the initiating node.

It should be noted, however, that individual cursor selections are not fully uncorrelated relative to the immediately preceding cursor selections. Over a long period, however, the selection does tend toward the required distribution (see Section 3.6.6).

There are two potential walks other than those mentioned above that could be used for random selection: the OnlyInLinks walk and the Metropolis-Hastings algorithm [68, 41]. OnlyInLinks can be expected to give good selection distribution on well-mixed graphs with degrees close to the desired degrees. While we do not evaluate OnlyInLinks as a selection walk in this chapter, we do so in Chapter 4, and find that it is an acceptable selection strategy when used on Swaplinks-built graphs.

The Metropolis-Hastings method is able to produce any desired selection distribution, but shares weaknesses with both the biased-halting methods and the biased-forwarding methods. Like SelfLoops, it incorporates self-loops in its operation (though not to the extent that SelfLoops does), so it is difficult to set the virtual hop-length to achieve a desired real hop-length. And like the inverse probability walks, it needs nodes to know the degrees of the neighbors, so would either need regular updates between neighbors, or run the risk of degraded performance. We therefore do not include this method in our evaluations.

3.5 Distributed Outdegree Computation

All the random walks proposed in the previous sections assume that nodes know their ideal outdegrees, where each node's ideal outdegree is proportional to its capacity. By ensuring that actual degrees and selection probabilities are proportional to their outdegrees, the walks in turn ensure that overall load at a node is proportional to its capacity. In this section we briefly outline how nodes can compute their ideal outdegrees.

We use the distributed scheme presented in [98] for the ideal degree computation. This scheme assumes that application load across the system is primarily along one dimension, e.g., bandwidth, CPU, disk, etc. The scheme also assumes that each node can gauge its own capacity to handle application-load along this dimension – this is the amount of resource that the node is willing to contribute to the system.

Once peers gauge their own capacities, they then need to determine their

ideal outdegrees such that each node's outdegree is proportional to its capacity. Since maintaining each new link itself leads to increase in the resultant load (due to neighbor heart-beats for example), we would like to minimize the degrees across the network while adhering to this requirement. For robustness of the random graphs, we also have the requirement that nodes have at least a certain minimum outdegree deg_{min} (of about 3) – this means that the node with the minimum capacity in the network should be assigned an outdegree of deg_{min}. Combining these conditions, we arrive at the following degree assignment, where deg is the outdegree being computed, c_{min} is the estimate of the minimum capacity across the network, and c is the capacity of the node doing the computation:

$$deg = \frac{deg_{min}}{c_{min}} \times c. \tag{3.1}$$

To estimate c_{min}, nodes periodically send out probes to randomly selected nodes; nodes that receive the probes respond with their capacity-values. Each node maintains a fixed-size window of (30-50 of) the most recently seen capacity-values. The straight-forward option for c_{min} is to set it to the minimum capacity value in the window. However, to avoid extreme outliers and to make sure all nodes arrive at about the same estimate of c_{min}, the 95^{th} percentile value of the capacity values could be used instead of the minimum.

Using the moving window lets the scheme naturally adapt to churn, since old samples are gradually phased out. Also, if the application frequently issues random selection requests, we could piggyback the probes on the selection messages, thereby almost completely avoiding any additional messages.

In the following evaluations, we assume that nodes have already correctly computed their ideal outdegrees. Our aim in the evaluations is to see how well the different strategies perform in getting the degree, selection probability, and message load distributions to align with the outdegree distribution.

3.6 Experimental Results

We start by describing the simulations used to evaluate the various approaches. We use static (non-time based) simulations. When simulating node additions or deletions, each node is fully added or deleted before the next node is added or deleted. Likewise, there is no notion of packet loss. While the simulations are not therefore fully realistic, we believe that they reflect the basic characteristics of the various approaches, and allow them to be legitimately compared. We believe this in part because of the random nature of our techniques—neither the order of events or the timing of events are very important.[8]

We examine two graph building scenarios:

(i) *Shrink:* A graph is built with a given number of nodes N – without any churn until all nodes have joined – and then nodes start leaving one at a time until the graph shrinks to 25% its original size
(ii) *Churn:* An N-node graph is built - without any churn until all nodes have joined - and then there are $2N$ churn-events, where a churn-event consists of either a single node kill or a single node join, with the same probability. The expected network size after this sequence of events is N. In all our measurements,

[8]We will however experimentally evaluate Swaplinks, the most attractive graph construction algorithm, and random selection over the Swaplinks graph in Chapter 4.

unless otherwise mentioned, we set N to 5000.

When the network only grows, i.e., when nodes only enter without leaving, Swaplinks' degree distribution (by design) is perfect, and therefore is not a fair comparison; we do not present these results here. On the other hand, the other schemes perform worse during the grow-only phase than they do under churn, because of the refreshing nature of churn (see below).

To measure the quality of random selection, we run $10.M$ selection walks using the algorithm to be evaluated, where the graph has M nodes at the time of selection (i.e., after the churn or shrink has completed), and look at the distribution of the selected nodes, and the selection load balance.

To model heterogeneity in our measurements, we use the following setting: Each of the N nodes in the graph is a *default-degree* node with probability 0.5, and a *heterogeneous* node with probability 0.5. Each default-degree node has an outdegree of 5. Each heterogeneous node chooses its outdegree uniformly randomly from the range [2,50]. As before, churn or shrink is performed on the graph after all nodes have joined and formed all their outlinks.[9]

The default setting we use in our experiments is N=5000 nodes, build-walk length of 10 hops, and, except in case of heterogeneity, a constant outdegree of 5 at every node. A walk-length of ten was chosen because this produced better results than a shorter walk-length, but longer walks did not perform significantly better than 10-hop walks. (In Section 3.6.5, we show that 10-hop build walks are sufficient for a wide range of network sizes.)

[9]By contrast, Gia simulated heterogeneity spanning three orders of magnitude. While indeed node capacities vary by this much in measured Gnutella networks, we do not believe that a node with 1000 times the capacity of a dial-up would be willing to devote all of that capacity to file sharing, and so use a more moderate capacity split.

For simplicity, we ensure that all build walks only find nodes that are not already neighbors of the initiating node, by storing the initiator's neighbor-list in the walks. This could be easily simulated in a real implementation by having the initiator retry if a build walk ended at a node that is already a neighbor.

Given that we have four graph-building techniques, four selection walks, heterogeneity, cursor walks, graphs of different sizes, and numerous parameters to measure, we need a way to prune down the results presented here. We do this by first evaluating the four graph construction techniques in terms of the "goodness" of the graphs they generate. We look at graph construction when all nodes have the same outdegrees, i.e., the *homogeneous* case in Section 3.6.1. We evaluate the performance of all the graph construction algorithms in conditions of heterogeneity in Section 3.6.2. Looking at these results, we pick the most promising graph building algorithm, which is Swaplinks, and present most of our subsequent results on that graph. We examine the quality of random selection: first we execute the four selection schemes over a homogeneous Swaplinks graph in Section 3.6.3, and then test all the selection walks over heterogeneous graphs in Section 3.6.4. We next look at the scaling behavior of the Swaplinks algorithm in Section 3.6.5. Finally, we evaluate the cursor mechanism in Section 3.6.6.

Table 3.2: Homogeneous graph construction: Degree distribution, diameter, and build-loads of the different mechanisms. All graphs except Scamp have exactly 5 outlinks per node, and use 10-hop build walks. *Diam* and *Dist* are the average estimated diameter and the average inter-node distance, estimated using a sample set of 20 nodes. *Dev(Deg)* is the standard deviation of degrees, *Indeg-95pc* is the average 95th percentile value, and *MaxIndeg* is the average maximum value of the indegree. *BLoad-Add* and *BLoad-Kill* are the loads caused due to node addition and node deletion, resp. (*)Scamp's *Indeg-95pc* and *MaxIndeg* values correspond to the total degree, since its outdegree is not a constant.

		Dev (Deg)	Indeg-95pc	MaxIndeg	Diam	Dist	Dev(BLoad-Add)	Dev(BLoad-Kill)	AvgBLoad-Add	AvgBLoad-Kill
Grow	TrueRandom	2.23	9.00	15.03	5.06	3.97	-	-	-	-
N=5K	Scamp*	6.97	28.24	44.68	5.34	3.45	7.81	-	10.60	-
	IP-Norefs	2.23	9.00	15.00	5.19	3.98	12.32	7.08	15.27	33.68
	IP-10refs	1.82	8.00	13.20	5.03	3.98	6.28	6.93	17.56	33.84
Churn	IS-Norefs	2.04	8.05	13.40	5.27	3.99	13.81	6.95	15.49	33.47
N=5K	IS-10refs	1.57	8.00	11.80	5.03	4.01	6.88	6.74	17.58	33.40
	SL-Norefs	2.03	8.00	13.34	5.30	4.00	5.54	5.36	9.51	14.87
	SL-10refs	1.55	8.00	11.66	5.03	3.99	4.60	4.63	9.58	12.58
	SW-NoRefs	1.31	7.00	11.66	5.01	3.99	4.11	5.19	9.63	17.86
	IP-Norefs	1.83	8.00	12.65	4.75	3.38	18.85	6.95	69.05	33.84
	IP-10refs	1.84	8.05	12.50	4.73	3.37	19.11	6.93	69.16	33.85
Shrink	IS-Norefs	1.58	8.00	11.10	4.77	3.38	21.18	6.70	70.75	33.31
N=5K to	IS-10refs	1.57	8.00	10.95	4.75	3.37	20.94	6.63	70.73	33.24
N=1.25 K	SL-Norefs	1.55	7.94	11.02	4.78	3.39	16.25	5.14	47.82	15.22
	SL-10refs	1.56	7.92	10.86	4.75	3.38	15.24	4.62	39.07	12.56
	SW-NoRefs	1.50	7.70	11.64	4.75	3.37	14.97	5.27	39.02	17.60
Piggy-	IP-Churn	5.31	12.15	75.45	5.02	3.86	16.66	11.84	6.61	10.98
back	IS-Churn	2.24	9.00	15.85	5.19	3.98	8.44	5.71	7.68	11.12
Only	IP-Shrink	2.74	9.50	27.70	4.83	3.38	18.40	4.62	32.53	11.18
NoRefs	IS-Shrink	1.85	8.00	12.90	4.74	3.38	20.16	4.87	34.93	11.15

3.6.1 Graph Construction (Homogeneous Case)

In this section we compare the different graph building algorithms in terms of the following parameters: degree distribution, network diameter and average distance between nodes, and distribution of the load placed on the network by the build walks. The graphs we study here are all homogeneous. We evaluate both graphs with and graphs without refreshes (except for Swaplinks, which does not benefit from refreshes). The refreshes are performed after the churn or shrink as described above has completed. For IS and IP graphs, we evaluate both the case where all immediate neighbors are informed immediately of any link change (*1-hop updates*) and the case where neighbor information is only piggy-backed on build walk messages (*Piggybacking*). We include in the comparison graphs built using Scamp, and *TrueRandom* graphs, where each node forms 5 outlinks with distinct uniformly chosen nodes in the network.

Ideally, at any given time, the load caused by the entry of new nodes or departure of existing nodes should be spread uniformly over the existing nodes in the network. We verify load balance under node addition in the following manner: 10 new nodes are added to the system and the load placed on previously existing nodes (barring the last 10 joiners[10]), in form of the number of messages received by them, is logged. This is repeated a total of 100 times with the load summed over the 100 times. Finally, the average load per node *AvgBLoad-Add* and standard deviation of the load values *Dev(BLoad-Add)* of all nodes is computed. We chose the comparatively small number of nodes added (10) here, as we want to focus on the load placed on already existing nodes: with increase in the number of nodes added, there is an increase in the load placed on the new

[10]The last 10 joiners would be unfairly heavily loaded because of the rendezvous scheme.

nodes themselves. Since this method of testing imposes the same load on the network irrespective of the size of the graph, the per-node load values are going to be higher for smaller graphs. To evaluate the load caused by node departures, we select $M/5$ nodes randomly, where the graph contains M nodes, and delete them (one by one) from the graph, and log the resulting load placed on the remaining nodes. We then compute the average load per node *AvgBLoad-Kill* and standard deviation of the load *Dev(BLoad-Kill)* caused by the deletions.

Table 3.2 shows the results for the homogeneous-capacity graph building simulations. A noticeable trend is that all parameters improve with refreshes, the improvement with a churned graph being more noticeable than that with a shrunk graph. This is because the effects of shrink ensures that each node will have refreshed its out-neighbor set multiple times with high probability, so a shrunk graph is effectively equivalent to a refreshed graph.

Another key thing to note from the results is that they are almost all reasonably good as far as the degree distribution is concerned. For instance, the standard deviation in node degree for TrueRandom is 2.23, and the only graph that did significantly worse than that was InlinkInvProb where neighbor information was only piggy-backed. Most did better than TrueRandom.

Swaplinks' policy of neighbor replacement ensures it has the best indegree distribution[11]. Swaplinks also has the best load distribution during node addition, mainly because its neighbor discovery walks use only inlinks and thus do not distinguish between nodes based on their degrees, since all nodes have the same outdegree. SelfLoops unfairly loads high-degree nodes because it does not bias among links during walk forwarding, while InlinkInvProb and Itera-

[11]In Swaplinks, the entry of new nodes negates, to a certain extent, the bad effects of prior node deletions, since each new node entry can only improve the degree distribution.

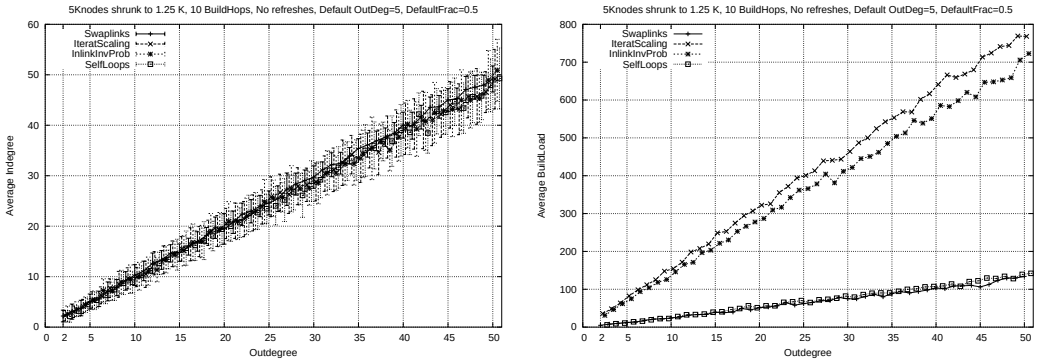
tive Scaling end up loading low-indegree nodes unfairly heavily as a result of their random walk weightings. InlinkInvProb and Iterative Scaling end up with high message load overheads anyway when they use 1-hop updates. The diameter and distance estimates are more or less the same for all the four building strategies.

The load during node deletion is the only parameter here that is worse for Swaplinks than for some of the other strategies. The reason here is Swaplinks' higher aggregate load during node deletions: neighbor discovery walks are initiated for in-neighbors as well as out-neighbors. Nevertheless, the Dev(BLoad-Kill) parameter with Swaplinks is still quite close to the other strategies. And, considering that neither refreshes nor neighbor information is required, Swaplinks may after all be more efficient as well as simpler.

Scamp here has the worst degree distribution, partially due to its larger average total degree of 15.7. We did not run churn or shrink on Scamp since Scamp does not explicitly discuss reacting to unannounced departures: Left unchecked, unannounced departures could lead to violating Scamp's desired property of node-degrees being logarithmic in the network size.

3.6.2 Graph Construction Under Heterogeneity

In this section we study how well the different schemes adapt to heterogeneity. The setting we will be using here is one where a 5000 node graph is shrunk or churned. Each of the 5000 nodes has, with a probability of 0.5 the default outdegree of 5, and with a probability of 0.5 a uniformly picked outdegree from the range [2,50]. We present results of the shrink case without refreshes; all the



(a) Avg. Indegree vs. Outdegree

(b) Avg. BldLoad-Add (Load due to node addition) vs. Outdegree

Figure 3.2: Heterogeneity : Variation of Build Parameters with Outdegree

other cases, namely, shrink with refreshes, churn with and without refreshes give similar results, which are not shown here. Graphs built using InlinkInvProb and Iterative Scaling make use of 1-hop updates.

We show here the average indegree and the build load during addition as a function of the outdegree. For each outdegree, we get the set of nodes with that outdegree, and compute averages from that set to get the figure for the particular outdegree. We use the same model to measure build load during node addition as we did in Section 3.6.1 (all 10 nodes added have degree 5). The distribution we want to achieve is one where all relevant parameters are directly proportional to the outdegree.

Fig. 3.2 shows the variation of the indegree and the build load during addition of new nodes. All strategies result in a linear dependence of both the indegree and the load on the outdegree, demonstrating that the modifications made to the walk probabilities indeed work as intended. In separate experiments, we found that the load during node deletion (not shown here) also grows linearly with the outdegree.

In the figure, the IS and IP load curves are much higher than the other two because of the 1-hop update load: each node that gains an inlink during the test needs to let all of its neighbors know, and with an expected total degree of 31 here, this results in a significant overhead. Note here that we could reduce the frequency of updates to achieve a smaller message overhead, but this comes at the cost of reduced accuracy of the maintained state. We do not evaluate this trade-off in this chapter. If we altogether drop the use of 1-hop updates with IS or IP, we will have to use proactive methods like planned refreshes, or exchange of neighbor information, or both, to generate good graphs; these result in overheads of their own.

Nevertheless, all build strategies do exhibit good control over heterogeneity, but we prefer the Swaplinks strategy over the others. There are two main reasons. The first is that it performs well along criteria such as degree distribution, diameter, load balance, etc. Second, and just as importantly, it seems the easiest to engineer: Swaplinks has just one parameter to set, namely the outdegree of each node[12]. With the other strategies, in addition to setting the outdegree, we need to worry about the frequency of exchanging neighbor information (with IP or IS), or about setting the virtual hop-length to achieve a target average hop-length (with SL), and the frequency of refreshing (IP, IS, SL). While none of these tasks is inherently difficult, it is nice to be able to avoid them since we can.

[12]Strictly speaking, all strategies need to also set the walk length to some value optimal to the number of nodes. Practically speaking, however, this can be set by default to a conservative large value such as 10 hops–see Section 3.6.5.

3.6.3 Quality of Random Selection on Homogeneous Graphs

Having picked Swaplinks as the most promising algorithm to build graphs (from Sections 3.6.1 and 3.6.2), we now evaluate the quality of random selection of the four selection schemes executing over a homogeneous Swaplinks graph. We use two parameters to measure the quality of selection: the distribution of the selected nodes, and the distribution of load imposed by the selection walks. The selection strategies TotalInvProb and Iterative Scaling make use of only piggybacked information sent over build walks, so these do not incur any extra message overhead to do state maintenance. We do not employ piggybacking on the selection walks here because the number of selection walks we use in the simulations is comparatively large, so piggybacking on even the selection walks would lead to an undesirable artificial improvement in the measured quality of selection.

We refer to the node selected by a random walk as the node *hit* by the walk. To evaluate selection quality, we start a set of random walks from a *single* node, and log the number of hits each node receives: we use a single start point to avoid the artificial smoothing introduced by having multiple start nodes. The number of walks executed is equal to 10 times the current number of nodes in the graph. We use the standard deviation of hits as the metric to measure selection quality.

We show the results of the shrink scenario here; results for the Swaplinks churn graph are similar. The results shown here correspond to a 5000 node graph before the shrink is performed. All nodes here have the same outdegree of 5.

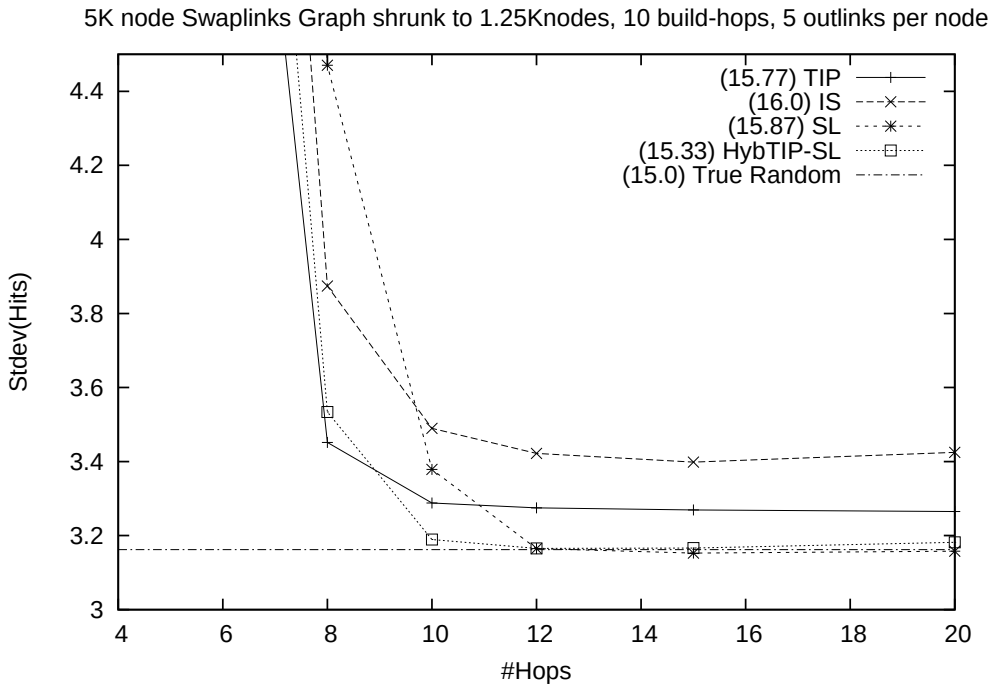


Figure 3.3: Std.dev(hits) vs. #Hops for the homogeneous Swaplinks graph. Numbers in parentheses indicate the 95th percentile value of hits at 10 hops (the average is 10 hits).

Fig. 3.3 shows the average standard deviation of hits as a function of the length of the walk. We use TrueRandom selection, where nodes are picked uniformly randomly from the entire population, as a benchmark. TrueRandom selection is just an instance of the balls-and-bins problem, resulting in a Poisson distribution of selection hits; its standard deviation of hits is given by the square root of the mean number of hits each node receives.

Once again, the main thing to note is that all of these walks perform satisfactorily well. The 95th percentile number of hits values are similar for all approaches, and not far from that of TrueRandom. Hyb-TIP-SL gives the best hit distribution on the Swaplinks graph, and this is very close to the best hit distribution using any mechanism on any other graph. TotalInvProb's selec-

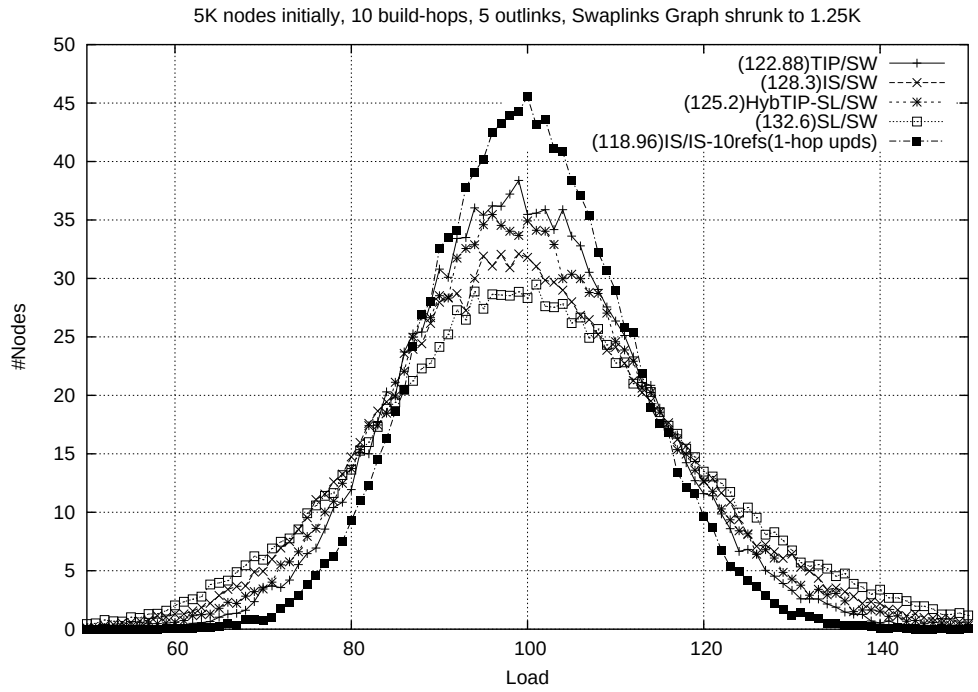


Figure 3.4: Homogeneous selection load distribution over the Swaplinks graph at 10 hops. In parentheses are the 95th percentile values of the load.

tion also is good, though it stabilizes at a distribution slightly different from TrueRandom's distribution. Iterative Scaling's distribution is not as good as the others because only piggybacking on the build walks is insufficient to bring the weights to the required state of convergence. Because SelfLoops is a variable walk-length strategy, its performance when the number of hops is small is poor since quite a few of its walks would be very short and end very close to the start point.

We measure the selection load seen by a node as the number of selection walks that pass through or end at the node. To measure the selection load, we again execute a given number of walks (again the number of walks is 10 times the number of nodes in the graph), this time with the origins of the walks distributed across the graph such that every node in the graph is selected as

the start node an equal number of times. The idea here is that since this is a homogeneous graph, the load distribution should be uniform when all nodes are involved in about the same number of walks – note that if some nodes start more of the walks than the others, there will be an unavoidable skew in the selection load seen by the nodes very close (within 3-4 hops) to the given start nodes. Fig 3.4 shows the bell curves of the selection load distribution when the walk-length is set at 10.

Note here that we have added one curve that is not based on the Swaplinks graphs: this is IS selection on an IS graph with neighbor information exchange and ten refreshes. We show this curve as a point of comparison because it is the best of all selection/build combinations. Among the remaining, TotalInvProb gives the best load distribution here, while Hyb-TIP-SL's selection load curve is slightly worse. Both of these curves themselves are reasonably close to the best (IS/IS) curve. Iterative Scaling as a selection mechanism on top of Swaplinks again suffers to some extent due to its imperfect piggybacked state. SelfLoops is the worst in terms of load-balance, as here the number of walks that pass through a node increases with its degree.

The decision of which algorithm to use to perform selection on the Swaplinks graph depends on the application. If each node performs selections relatively infrequently, then the algorithm to use would be TotalInvProb (since Hyb-TIP-SL has the virtual walk-length selection problem), or even OnlyInLinks, which we evaluate in the next chapter and show is a good selection mechanism. If, on the other hand, selection walks are very frequent, then Iterative Scaling might be the strategy to use, because it can converge to the required state via piggybacking on top of the selection walks.

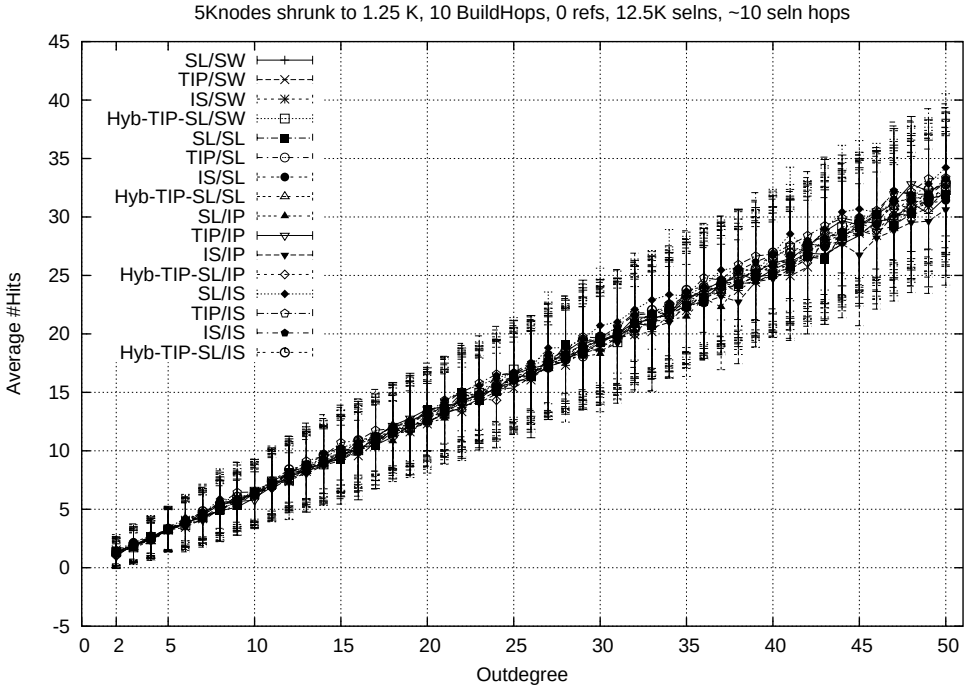


Figure 3.5: Heterogeneity: Average Hits vs. Outdegree

3.6.4 Selection with Heterogeneity

We now look at the quality of selection when nodes have different outdegrees. We use the same setting we used for evaluating graph building under heterogeneity (section 3.6.2), i.e, a 5000 node graph subjected to shrink, and the same expected outdegree distribution. We present the results of running 12,500 random selection walks using each of the four selection algorithms on top of all the four different graphs. We measure the distribution of selection hits as a function of the outdegree.

Fig 3.5 contains the results. The selection hits vary linearly with outdegree for all combinations of selection strategies and build methods. The selection load curve (not shown here) follows a similar pattern: linear, with even smaller variance. Thus, we are able to engineer all of the selection methods to function

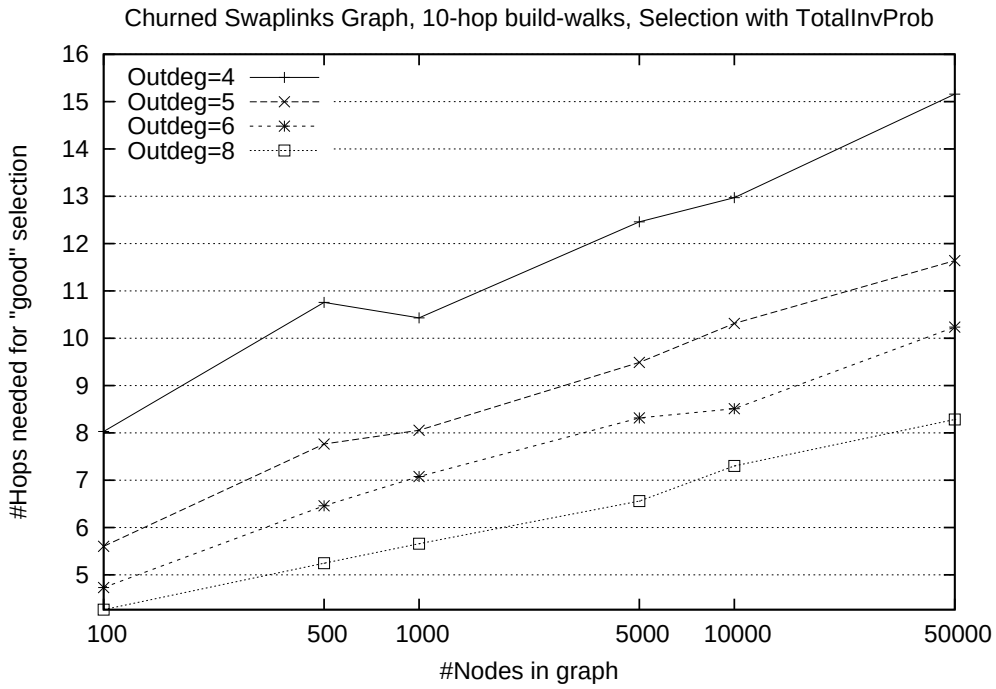


Figure 3.6: Variation of the required selection walk-length for a range of network size and average degrees

satisfactorily well with regard to heterogeneity.

3.6.5 Scaling to Larger Sizes

In this section we evaluate the scaling behavior of Swaplinks over a wide range of network sizes and average degrees: we vary the network size from 100 to 50,000, and the outdegree per node from 3 to 8, and measure the number of hops it takes to obtain a random selection distribution whose standard deviation is within 5% of that of true random distribution. The graphs are churned before the selections are performed; the number of selections is ten times the network size. We use TotalInvProb as the selection mechanism here. All build walks are 10 hops in length. Fig. 3.6 shows the results.

Table 3.3: Graph parameters for 50,000 node churned graphs.

Deg	Dev(Deg)	95pc(Deg)	MaxDeg	Diam	Dist
4	1.22	6.0	11.0	7.0	5.65
5	1.32	7.0	14.0	6.15	4.93
6	1.39	8.0	15.0	6.0	4.63
8	1.52	11.0	16.0	5.05	4.13

With only 3 outlinks per node, TotalInvProb was not able to consistently reach the required quality of selection when the system size grew beyond 1000, so these results are not shown. When the outdegree is more than 3 though, TotalInvProb reaches the desired quality. The number of hops needed grows logarithmically with the network size, and, as can be expected, decreases as the average degree increases. The rate of change of the number of required hops as the system size increases is very small. From a practical perspective, this would allow someone deploying a P2P application to select a conservative but reasonable value for number of hops given their largest expected user population.

To verify that Swaplinks builds good graphs even at large scale, we show in Table 3.3 the indegree distribution and the estimated diameter and average distance for 50,000 node churned graphs for different values of outdegree per node. These results show that the graph building mechanism and the selection walk procedures both scale well. In addition, these results, along with those presented in the previous subsections, also demonstrate the robustness of our methods to the kinds of network churn tested in this chapter.

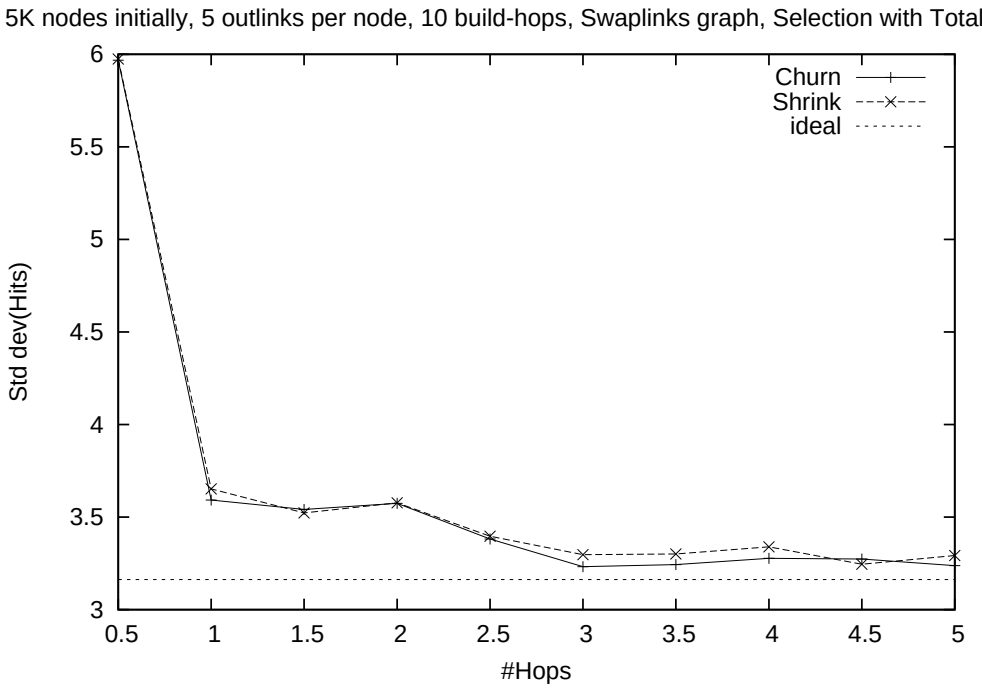


Figure 3.7: Std Dev(Hits) vs. #Hops in each Cursor Walk

3.6.6 The Cursor Approach

In this section, we evaluate the cursor walk described in Section 3.4. Fig 3.7 shows the variation of the quality of hit distribution with increase in the expected walk-length[13]. The total number of cursor walks initiated here is equal to ten times the network size. The result shows that the approach is indeed viable, with about 3 hops on each small walk needed to approach the uniform distribution. When the walks are shorter than this length, the probability of reselecting already selected nodes increases, affecting the selection distribution. A trend that can be noticed is that even numbered hops are local maxima in the plot. We believe this is because with an even hop length, the probability of the walk backtracking and returning to the previously selected node increases.

[13]Here a fractional walk-length of say 1.5 hops corresponds to the set of cursor random walks where each walk is independently of length 1 or 2, with probability 0.5 each.

3.7 Conclusions and Discussion

The broad conclusion that we draw from this work is that our original goal – to find a simple and scalable mechanism for building random graphs and doing random selection, with good control over heterogeneity – is certainly satisfied. Specifically, our Swaplinks approach lets us construct graphs while requiring the setting of only a single parameter by each node, namely its desired node degree, and enables the desired random selection on top of the graphs thus built. We were also able to adapt the other algorithms we studied, namely the inverse-probability walks, Iterative Scaling, and SelfLoops, to the requirements of heterogeneity, but Swaplinks is comparable to or better than these in terms of performance, while being much simpler to deploy.

The next step is to implement the Swaplinks algorithm, and test it in a real setting (i.e., Planetlab), which we do in the next chapter. We also compare this unstructured strategy with a random selection strategy that uses DHTs.

As described, all strategies we developed or adapted in this chapter have a few weaknesses: They are all vulnerable to misbehaving nodes. Although not reported, we ran experiments with the biased-walk approaches where misbehaving nodes would terminate every build walk at themselves. Even without creating any additional outlinks, these nodes obtained inlinks with almost every every other node in the graph![14] We need to explore simple mechanisms to prevent this.

Another point to consider is that many applications benefit from being able to find proximal neighbors (those with low latency), in addition to random

[14]Note that this attack is different from one where a misbehaving node simply sets its desired degree to a very high value, thereby legitimately collecting many links, which it can then abuse.

neighbors; all of our strategies in this chapter however select purely random peers. While finding proximal neighbors could in theory be left to the application (once it has a random network to "explore"), it seems that providing this capability as part of a single set of mechanisms would be broadly useful. We will study the problem of finding proximal neighbors in Chapter 5.

3.8 Acknowledgements

We would like to thank Robbert VanRenesse, Emin Gun Sirer, and Jon Kleinberg for their participation and help early in this project.

CHAPTER 4

COMPARISON OF STRUCTURED AND UNSTRUCTURED APPROACHES TO HETEROGENEOUS PEER SELECTION

4.1 Introduction

In the previous chapter, we examined the problems of heterogeneous random graph construction and heterogeneous peer selection in p2p networks. Using simulations, we evaluated multiple unstructured approaches to solve these problems, and identified the Swaplinks algorithm as the most attractive solution. We examined only unstructured approaches because the applications we were interested in were all unstructured, and because of our intuition that unstructured approaches would be simpler than structured approaches, and that this simplicity would ultimately lead to a more scalable and robust system.

In this chapter, we focus on the problem of heterogeneous peer selection, and verify the above intuition: We want to answer the question of whether heterogeneous peer selection is best performed by an unstructured approach or by a structured approach. We perform a thorough performance comparison of a structured heterogeneous selection approach with that of Swaplinks. Heterogeneous peer selection is a significant enough problem to warrant this attention: As noted in the previous chapter, heterogeneous peer selection is an important functionality required by many diverse unstructured p2p applications, like file-sharing applications (Gnutella, [1, 64, 14]), overlay multicast applications ([104, 44, 26, 56]), proximity systems ([19, 17]), etc. We do not study the related question of which of unstructured and structured approaches are best- suited to perform random graph construction, as this is not as interesting: a random

graph is an unstructured graph, so unstructured approaches are bound to be more appropriate here.

We implement Swaplinks and use the implementation in the above performance comparison. In the process, we also provide a comprehensive evaluation of the Swaplinks implementation, thus validating simulation results from the previous chapter.

For the structured random selection approach, we adapt the "item-balancing" algorithm for load balancing in structured P2P networks by Karger and Ruhl [51, 50]. Please see Section 2.2 for why we picked this approach over other competing approaches. The basic idea in using the item-balancing algorithm in our setting is to assign identifiers in the DHT number space such that a larger portion of the number space maps proportionally into high-capacity nodes, and a smaller portion maps into low-capacity nodes. High capacity nodes, by virtue of "owning" a larger portion of the number space, will be selected proportionally more often by queries issued to uniformly random identifiers. We implement the Karger/Ruhl approach on the Bamboo DHT [83], and call this approach KRB. We chose Bamboo because it is a stable well-maintained open software for DHTs, and because it is a second generation DHT, designed using the best principles from the earlier, first generation DHTs (like Chord [99] and Pastry [86]). This minimizes the chances that the results are an artifact of a poor DHT implementation.

The performance comparison between KRB and Swaplinks shows that KRB performs less well in the face of churn, and has a number of hard-to-set tuning knobs that affect performance. While we need more comparisons (with other structured approaches) to be certain of this, the results of the evaluation agree

with our intuitive concern about the relative complexity of using DHTs for heterogeneous peer selection.

Overall, we make two contributions in this chapter:

- We implement Swaplinks, and experimentally measure its performance for both random graph construction and random selection, and in so doing validate earlier simulation results.
- We modify the Karger/Ruhl load balancing algorithm for heterogeneous random peer selection, and compare its performance as a random selection mechanism with that of Swaplinks.

We next describe the Swaplinks implementation in Sections 4.2, and the KRB method in Section 4.3. In Section 4.4 we give a performance evaluation and comparison of both algorithms. Finally, we discuss issues and future work in Section 4.5.

4.2 Swaplinks Implementation

We implement Swaplinks in C++ on Linux. We use TCP sockets for neighbor connections. Each node sends heart-beat messages to each of its neighbors every 2 seconds, and assumes that a neighbor is dead if it does not receive a heart-beat from it for 10 seconds. [1]

A newly entering node initiates the required number of neighbor discovery

[1]For the results shown in this chapter, we do not utilize the TCP socket close signal as an indicator of neighbor departure, so as to have a fair comparison with KRB, since Bamboo uses UDP

walks, restricting the number of outstanding neighbor walks to 10 at any time. A neighbor discovery walk is re-attempted if it fails to return an appropriate neighbor within a period of 2 seconds.

We also implementat the rendezvous server as specified in the previous chapter to help new nodes join the system. The rendezvous server remembers a small number (currently 10) of the most recently joined nodes, and newly joining nodes use these nodes to start their neighbor discovery walks. This rendezvous mechanism is light-weight, and makes sure no single node is overloaded with the responsibility of helping new nodes join the network. The rendezvous mechanism could be made more robust by also having the rendezvous server remember a small number of random other nodes in the network, by periodically taking random walks, or by having newly joined nodes report one or two of their neighbors.

In our implementation, we use OnlyInLinks for application-requested peer selection, rather than the other random walks tested in the previous chapter. While both OnlyInLinks and the random selection walks studied earlier result in selection proportional to nodes' outdegrees, OnlyInLinks is simpler and as we find, results in acceptable selection properties.

The application using Swaplinks communicates with the Swaplinks module via a TCP socket. One application has currently been implemented over Swaplinks, namely a heterogeneous overlay multicast protocol called ChunkySpread [104] that uses Swaplinks to both construct a heterogeneous random graph and do random peer selection. Each Chunkyspread node is involved in multicast data transmission (and reception) with multiple other nodes; this set of peers is a subset of the set of the neighbors in the Swaplinks graph. A

small set of ChunkySpread nodes (the nodes that originate the multicast stream) need to discover an additional set of peers. This is done using Swaplinks peer selection. In addition to ChunkySpread, the Swaplinks algorithm has been used in other applications, like an extension of the Stunt toolkit for NAT traversal in P2P systems [37], and an experimental P2P file backup system.

We are also currently experimenting with an alternate heart-beat mechanism, called *smart-pinging*, which reduces heart-beat load at nodes with very high degrees. We describe smart-pinging and give a preliminary evaluation of the technique in Section 4.4.4.

4.3 Adapting the Bamboo DHT to Heterogeneity

4.3.1 The Bamboo Distributed Hash Table

Distributed Hash Tables (DHTs), as the name suggests, distribute the functionality of a hash table across many different nodes. Example DHTs include Chord [99], CAN [80], Pastry [86], Tapestry [42], Bamboo [83], and others. Most DHTs work with keys chosen from a finite unidimensional identifier-space (ID-space) that wraps around itself. The ID-space is sometimes referred to as a "ring" because of the wrap-around feature. Nodes themselves are assigned identifiers (IDs) chosen from the same space. DHTs map each key in the ID-space to a unique node in the population. This mapping also enables the ID-space to be split among the different nodes, where different nodes are "in charge" of different parts of the space. The key-to-node mapping is determined by the key and the IDs of the nodes. In Chord for example, a key is mapped

to the *predecessor* node of the key, i.e., the node with the largest ID not larger than the key (with wrap-around if necessary). In Pastry and Bamboo, a key is mapped to the node with ID numerically closest to the key.

Given any ID, DHTs also provide the functionality of efficient overlay routing from any member of the DHT to the node in charge of the ID.

The Bamboo DHT uses two sets of neighbors, namely the *leaf set* and the *routing table*, to enable routing to a given ID, and to tolerate the possibility of nodes leaving the system. It uses periodic recovery, where it repairs failed neighbor entries at a fixed rate, rather than reactive recovery, where a discovery of a failed neighbor immediately triggers a repair. It also incorporates features to make sure the overlay hops are between nearby nodes (proximity neighbor selection), and an adaptive mechanism to compute acceptable neighbor-timeout values. Please see [83] for the details.

4.3.2 KRB

Performing random selection on a DHT, with no regards to heterogeneity, and assuming the ID space is apportioned uniformly among all nodes, is simple: pick a uniformly random ID in the ID space, issue a random selection query to that ID, and select the node where the query ends. For this simple querying mechanism to still be applicable when there are differences in node capacities, we need to split nodes' ID spaces in proportion to their capacities (where a "node's ID space" denotes the extent of ID space that the node owns). For a simpler design of the heterogeneous random selection scheme, we choose to compute a node's ID space as the space between its successor in the ring and

itself. We discuss how we simulate this feature in Bamboo later in this section.

To achieve capacity-dependent ID space allocation, we develop a scheme based on the item-balancing algorithm (henceforth referred to as *K-R*) presented by Karger and Ruhl in [51, 50]. Nodes in K-R periodically send messages to one another, and share loads when a load imbalance is perceived. The item-balancing algorithm in [51] performs load sharing through movement of nodes to new IDs, but does not address the issue of heterogeneity, whereas the one in [50] takes heterogeneity into account, but does load sharing by transferring *items* from heavily loaded nodes to lightly loaded ones. Our scenario is slightly different from either of the above two, since we need nodes to move to new IDs so as to do ID space partitioning, and we need this partitioning to be sensitive to differences in capacities.

We now outline KRB, our adaptation of the K-R algorithm. The basic aim of KRB is to even out the *relative* loads of all nodes, where a node's relative load is its ID space load divided by its capacity. As in K-R, each node periodically sends out a message to a randomly chosen ID, embedding its load information – we call such messages "KRB load messages". Noting that a node's moving to a new ID can affect the ID spaces of (up to) 3 nodes (the moving node, the moving node's old predecessor, and the moving node's new predecessor), in KRB, we examine the change in load at *all* nodes whose loads are affected by the move. This is an extension of K-R, where the loads at only the moving node and the moving node's new predecessor are examined. If we examined the loads at only these two nodes, it would be possible for a huge load to be inadvertently dumped on the unconsidered third node (the moving node's old predecessor) as a result of the move; by considering all the three nodes, we avoid this possibility.

Looking at a single KRB load message, let us denote by S the node that sent out the message, by R the node that receives the message, and by P the predecessor of R. Now R decides if it should move to share S's ID-space, based on the value of the *objective function*, computed as follows:

$$r = \frac{L_R + L_S + L_P}{C_R + C_S + C_P}$$

$$ObjFn(R, S, P) = \Sigma_{N \in \{R,S,P\}} \left| \frac{L_N}{C_N} - r \right| \tag{4.1}$$

where L_N is node N's ID-space load, which is equal to the space between N and its successor, and C_N is node N's capacity.

If R were to move, it would move to ID R' such that

$$L_{R'} = \frac{L_S \cdot C_R}{C_R + C_S} \tag{4.2}$$

That is, the new ID is the one that splits the space between S and its successor in direct proportion to their capacities. If R were to move to R', the objective function would take on a new value, computed similarly to above. Finally, R does make the move if the objective function value reduces by more than a threshold ratio (called the *KRB-threshold*, set to 0.2).

The computation of the objective function above can be seen as a greedy step taken towards minimizing the system-wide objective function, given below.

$$r_{all} = \frac{\Sigma_{N \ in \ system}(L_N)}{\Sigma_{N \ in \ system}(C_N)}$$

$$ObjFn(overall) = \Sigma_{N \ in \ system} \left| \frac{L_N}{C_N} - r_{all} \right| \tag{4.3}$$

Since individual nodes do not know the value of r_{all}, they use local knowledge to compute r as shown above as an estimate.

The above description assumes that the node that sent the initial message S is not already the predecessor of the node that receives the message R. If R does happen to be the successor of S, there is no other third node whose load will be affected if R were to move to any point in between S and its successor. So now R moves if the following condition holds:

$$\frac{L_S}{C_S} < \epsilon \frac{L_R}{C_R} \quad OR \quad \frac{L_R}{C_R} < \epsilon \frac{L_S}{C_S}$$

where we set ϵ to 0.8. This criterion is identical to the one used in K-R.

Simulating a node's ID-space in Bamboo:
To make this scheme work in Bamboo, we need to make sure that the probability that a node is selected is proportional to the ID space for which it is the closest predecessor. However, in Bamboo, a query is routed to the node numerically closest to the destination, rather than to the closest predecessor. So when a node receives a random selection query, it examines the intended destination ID and forwards it to the immediate predecessor of that ID.

Our primary goal in adapting the Karger/Ruhl scheme to Bamboo was capacity-sensitive random peer selection. Admittedly, this scheme does not balance message load according to capacities (during the construction of the KRB network or during random selection), as we only tailor nodes' ID spaces, and not their routing tables. Accordingly, in this chapter, we evaluate KRB as a heterogeneous selection mechanism alone, and do not place emphasis on the message load distribution that occurs while constructing the KRB P2P network. Schemes that reactively tailor the neighborhood size based on capacity, such as those proposed in Accordion [60] and HeteroPastry [12] could be used with KRB to achieve both capacity-sensitive probability of selection and capacity-sensitive message load distribution during graph construction.

4.4 Performance Evaluation

We test Swaplinks through an emulation of a 1000 node network on either a local (Cornell) cluster of 5 machines with 4 CPU's each, or a 20 CPU cluster on Emulab. We achieve this size by launching a number of processes that in turn launch the required number of individual instances of our system. We preserve the semantics of communication here: all communication still takes place through sockets. The CPU loads here were mostly small enough to be negligible as a factor in the results. We also test the same implementation on PlanetLab.

For the emulation, we use a Transit-stub [112] topology consisting of 100 routers to mimic latencies between peers. Each peer picks a stub router uniformly at random. All messages to be sent are buffered at the sender for the appropriate amount of time (computed as a function of the stub routers of the source and destination). We also add jitter as a random value that ranges between 0 and 25% of the end-to-end latency.

Launching KRB networks of a similar size (500-1000 nodes) by multiplexing several instances on single hosts on local clusters proved infeasible because of high CPU load factors due to the Bamboo implementation. We instead evaluate KRB using the simulator available with Bamboo's standard code distribution. We use the same Transit-Stub topology as earlier to calculate message delays in the KRB network. We had to restrict our comparisons to 1000 node networks as the Bamboo simulator, with our modifications, consumes too much memory for larger sizes.

All newly joining KRB nodes contact a single *gateway* node, as Bamboo calls

them, that helps them enter the overlay. A node that leaves its present spot and rejoins the system as part of the KRB ID space readjustment scheme uses the set of neighbors it had before it left the system as its gateway nodes.

We now give a road map of the experimental results that we will be presenting in the subsequent portions of the chapter. We first test 1000 node networks of both Swaplinks and KRB under two different representative values of churn, and, similarly under two different distributions of node capacities (Section 4.4.1). Next, we subject both to more demanding churn scenarios: one where network size doubles in the space of 10 seconds, and one where network size halves instantaneously (Section 4.4.2). We give results of a 250-node experiment over planetlab in Section 4.4.3. Finally, in Section 4.4.4 we describe how we can use "smart-pinging" to reduce the heart-beat load incurred by high degree nodes in Swaplinks.

In all of these experiments, we evaluate Swaplinks as both a heterogeneous graph construction mechanism (e.g., how well node degrees match desired degrees) and as a heterogeneous peer selection mechanism (e.g. how close the selection probabilities are to the desired values). We evaluate KRB on the other hand as solely a heterogeneous peer selection mechanism.

4.4.1 Evaluation under representative churn scenarios

We use two separate churn scenarios: a "high-churn" scenario in which the median session time is 2 minutes , and a "low-churn" scenario in which the median session time is 30 minutes. These session time values have been taken from previous studies [90, 89, 38]. We similarly use two capacity distributions:

(i) The first capacity distribution is a 'moderate' 5:10:20 distribution, with 80% of Swaplinks nodes having outdegree 5, 10% having outdegree 10, and 10% with outdegree 20. We realize the same (relative) capacity split in KRB by having 80% of the nodes have a capacity of 1, 10% have a capacity of 2, and 10% of the nodes have a capacity of 4. (ii) The second capacity distribution is an 'extreme' 3:60:150 distribution, with 98% of the nodes with outdegree 3, 1% of the nodes with outdegree 60, and 1% of the nodes with outdegree 150. Again, we realize the same relative capacity distribution in KRB as well. We restrict the number of high-capacity nodes in the extreme capacity distribution to the relatively small proportion of 1% for the following reason: We run most of our experiments on networks of size 1000. With an increase in the number of high-capacity nodes, it gets more likely that there is a completely connected 'core' made of the high-capacity nodes, and all other nodes directly connected to the core nodes. The fact that this behavior is not retained when the network grows to a larger size (where the network maintains the same capacity distribution) makes such networks not representative of general P2P settings.

We use node session times that are independent of capacities, and follow the Pareto distribution. Networks start from scratch (zero nodes), and total experiment times are typically set to more than 5 times the median node session times. We ran tests where node session times were dependent on capacities (i.e., where high capacity nodes are likely to stay in the system longer) and where session times were Poisson distributed, and we found the results to be similar to those we present here.

Unless otherwise mentioned, we run ongoing background peer selections, where the 80 longest living nodes perform a random selection every 250 ms for

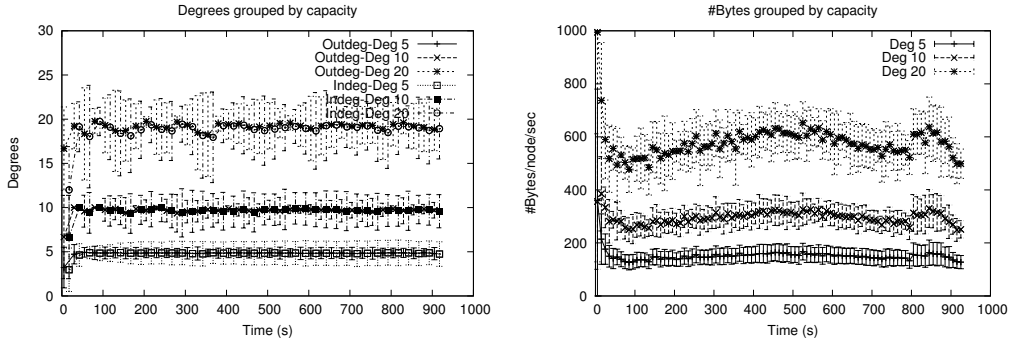
the duration of their lifetime. We call such selections 'periodic' selections. We use the periodic selections to evaluate whether, for instance, the degree 20 nodes receive, on average, twice as many selections as do degree 10 nodes, over the course of the experiment. We also have two other nodes perform a 'burst' of 10,000 selections with a gap of 10 ms between successive selections. We use the short-term burst to obtain a set of selection measurements with relatively little churn. This allows us to more accurately compare the measured distribution of selections among a group of same-capacity nodes with the ideal distribution. This is because each node present in the network during the burst receives a statistically large number of (measured or ideal) selections. The burst selections are performed just before the end of each experiment.

We measure message loads in both Swaplinks and KRB by counting only the bytes in the message payloads; we do not consider TCP/IP or UDP header overheads.

Swaplinks Results

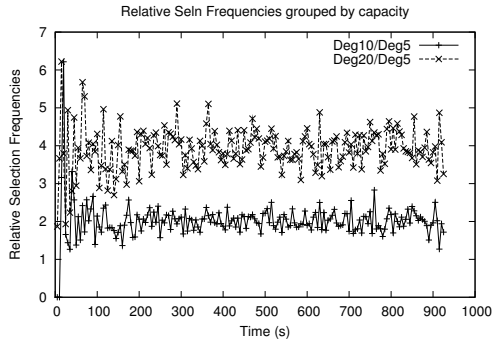
Figures 4.1 and 4.2 show the results of the high-churn, moderate capacity distribution experiment for Swaplinks. The node degrees closely track the desired values (Figure 4.1(a)), while the selections and message loads are split among the different nodes in proportion to their capacities: for example, nodes with outdegree 10 receive twice as many selections, on an average, as the nodes with outdegree 5. Both periodic and burst selections are counted to compute the curves in Figure 4.1(c).

Figure 4.2 shows the selection frequencies that result from the burst selec-



(a) Degree Distribution

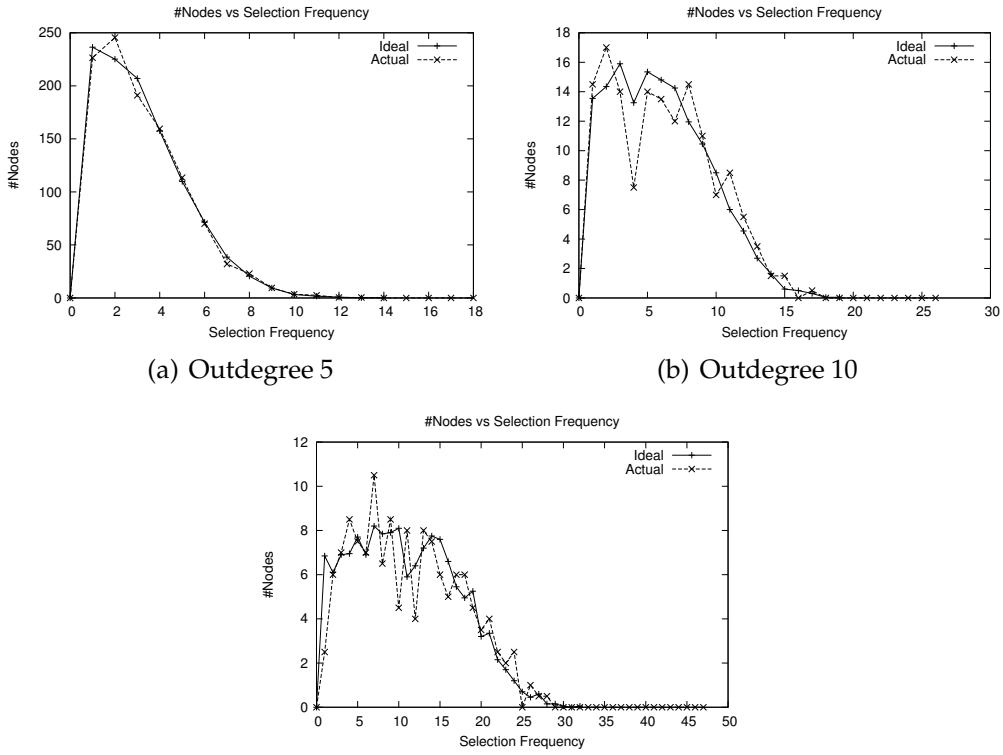
(b) Message Loads



(c) Relative Selection Frequencies

Figure 4.1: Swaplinks under high churn and moderate capacity distribution

tions. The figure has one plot for each of the three different capacity classes, where a capacity class is just a set of nodes with the same capacity. The "actual" curve represents the Swaplinks selections. The "ideal" curve represents the ideal distribution of the particular class' 'fair share' of the total number of successful selections; the intersection of each node's lifetime with the timespan of the burst selections is taken into account in computing this distribution. These values don't include failed selections, which occur with churn because nodes take about 10 seconds to detect that a neighbor is down. Thus, the higher the churn-rate is, the greater the probability is that a selection walk fails by being forwarded to a now-dead neighbor at some hop. High churn has about 40%-45% failed selection walks, while low churn has about 2% failed walks.



(a) Outdegree 5

(b) Outdegree 10

(c) Outdegree 20

Figure 4.2: Swaplinks #Nodes vs Selection Frequency for each degree: high churn and moderate capacity distribution

As can be seen from the plots in Figure 4.2, Swaplinks' actual selection frequency distribution closely tracks the ideal curve for each of the different capacities. This, coupled with the fact that Swaplinks also realizes capacity-wise selection distribution (Figure 4.1(c)), demonstrates that the selection mechanism realizes the desired distribution.

Table 4.1 gives a summary of results from all of the Swaplinks experiments in this section by averaging each value over the second half of the experiment time. The duration of high-churn experiments here is around 930 seconds, whereas the duration of the low-churn experiments is around 14,000 seconds. Each row in the table corresponding to a 'relative' value shows the corresponding value

Table 4.1: Swaplinks results for moderate and extreme capacity distributions under high and low churn.

	High Churn					
Target Outdeg	5	10	20	3	60	150
Avg Load(B/s)	150.26	297.87	581.49	98.16	1621.23	3449.41
Relative Load	1	1.98	3.86	1	16.43	35.05
Avg Totaldeg	9.68	19.41	38.23	5.80	111.77	255.25
Relative Selns	1	2.01	3.95	1	19.78	44.43
Seln p-values	0.815	0.862	0.977	0.757	0.579	0.819
	Low Churn					
Target Outdeg	5	10	20	3	60	150
Avg Load(B/s)	124.33	247.34	491.49	79.62	1275.89	2903.17
Relative Load	1	1.98	3.95	1	15.92	36.20
Avg Totaldeg	9.98	19.93	39.95	5.98	119.98	298.18
Relative Selns	1	2.00	3.99	1	20.10	50.16
Seln p-values	0.292	0.784	0.583	0.224	0.957	NaN

for the capacity class as a ratio over the equivalent value in the lowest capacity class in the experiment. Both periodic selections and burst selections are taken into account in computing the 'Relative-Selns' row. The last row plots the χ^2-test p-values of the selection frequency distribution (for burst selections): this is an indicator of how well the actual selection frequencies of nodes within each capacity class match the ideal selection frequencies. Larger values indicate a closer match; p-values greater than 0.05 are generally believed to indicate a good match of the observed distribution with the expected distribution.

As can be seen from the table, with the one exception of the high-churn

extreme-capacity case, node degrees and selection frequencies closely track the desired values. The valid p-values are all comfortably greater than 0.05, indicating good selection distribution.[2]

In the high-churn extreme capacity case, high degree nodes have an average total degree that is less than the respective ideal values: High degree nodes need some time to reach their full degrees upon entering the system, because they have at most 10 neighbor discovery walks outstanding at any time. This effect is more prominent during high-churn, where new nodes enter more frequently. The values for the relative selection frequencies suffer because of the imperfect degree distribution, but they nevertheless are still reasonably close to the target ratios.

The message load ratios in the extreme capacity distribution deviates from the ideal 3:60:150; this is because some of the high-degree neighbors have duplicate links between them, resulting in a reduction of the heart-beat load incurred. This is an artifact of the fact that the total degree of the highest capacity nodes here is non-negligible in comparison to the total number of links in the system, and we expect the number of duplicate links to decrease and the load-ratios to get closer to the 3:60:150 proportion in larger networks.

Looking at the message loads from an alternate perspective, the absolute values of the message loads for the outdegree 60 and outdegree 150 nodes seem relatively high. The bulk of this load is caused by neighbor heart-beats. In Section 4.4.4 we describe how we can reduce this load by using heart-beats in a more sophisticated fashion.

[2]The single "NaN" entry indicates that there were too few (<5) nodes of the particular capacity during the time when the burst selections were performed for a meaningful p-value to be computed

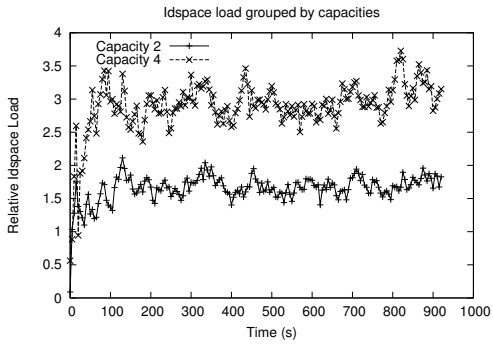
Table 4.2: Modification of various timeout parameters according to churn settings. "Original" denotes the values in the original Bamboo code distribution. The first four parameters determine the frequency of pings and exchanges of neighbor-sets between different nodes. †*discard_nbr_timeout* denotes the time between when a Bamboo node suspects a neighbor to be down (due to failure of message delivery) and when it actually decides it's down (due to lack of response to subsequent pings). ‡*KRB-period* is the period between successive KRB load messages sent to random locations in the network.

Parameter	Original	High-Churn	Low-Churn
`periodic_ping_period`	20 s	1 s	2 s
`ls_alarm_period`	4 s	1 s	3 s
`near_rt_alarm_period`	10 s	2 s	7 s
`far_rt_alarm_period`	20 s	5 s	15 s
discard_nbr_timeout†	60 s	1 s	1 s
KRB-period‡	–	5 s	10 s

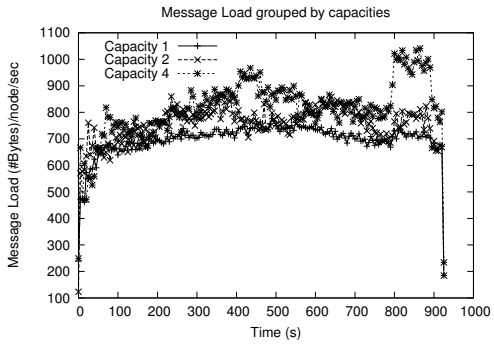
We ran similar experiments for 5000 nodes Swaplinks graphs over a 20 CPU cluster on the Emulab testbed, and found the results to be broadly similar, demonstrating that Swaplinks retains its properties in larger networks as well.

KRB Results

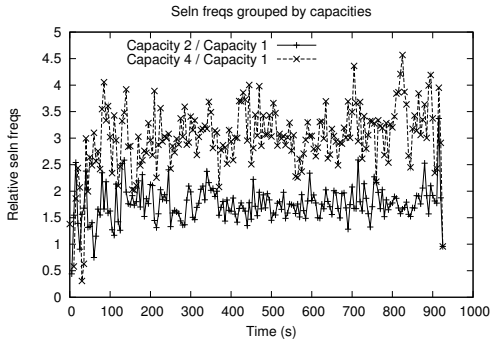
We use the Bamboo code released on July 1st 2005 for the KRB simulations [4]. We make quite a few changes to the parameters used by the default Bamboo source distribution to get KRB to approach the desired relative capacity-wise ID space distributions (Table 4.2). The Bamboo paper [83] cautions against using small timeout values, lest phantom neighbor failures are detected. While we do reduce the timeout values from the orginal Bamboo source distribution, since



(a) Relative Id Space Distribution



(b) Message Loads

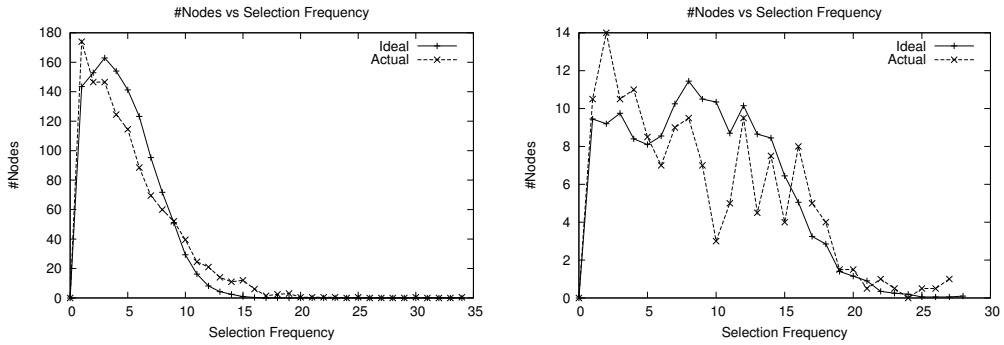


(c) Relative Selection Frequencies

Figure 4.3: KRB under high churn and moderate capacity distribution

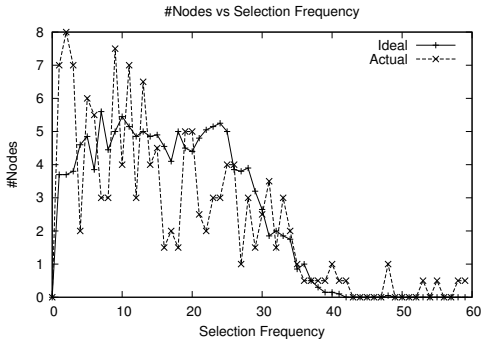
we use simulations and do not impose any bandwidth limits, phantom failures are not an issue here. We use a leaf-set size of 4 and a KRB-threshold value (see Section 4.3) of 0.2 in all KRB simulations. In the simulations, we do not use the coordinate-based timeout calculations provided by Bamboo (see [83]), since latency is not of concern in the evaluations in this chapter. We restricted KRB low-churn simulations to a shorter duration of 1800 seconds; longer simulations took unreasonably longer (wall-clock) times to complete.

Figures 4.3 and 4.4, and Table 4.3 show the results for KRB. The results show that KRB is not successful in maintaining the relative ID-spaces at the desired levels under high churns – it is only able to achieve around a 1:1.65:3 relative division in the ID spaces in the moderate capacity distribution, while its response



(a) Capacity 1

(b) Capacity 2



(c) Capacity 4

Figure 4.4: KRB #Nodes vs Selection Frequency for burst selections: high churn and moderate capacity distribution

to the extreme capacity distribution under high churn is worse. KRB is able to achieve the desired relative ID-space distribution in the low-churn moderate-capacity case, but again fails to fully achieve the desired ID-space distribution in the extreme-capacity low-churn scenario. KRB also fails to consistently achieve the desired selection distribution within each capacity class, as seen by the burst selection p-values computed for the selection frequencies. The p-values for the lowest capacity class in the moderate capacity distribution is 0 in both the high and low churn scenarios. This is mainly because the actual selection distributions here have quite a few outliers – nodes with an actual selection frequency close to or greater than the maximum selection frequency (for any node) predicted by the ideal curve. KRB's selection frequency curves within capacity

Table 4.3: KRB results for moderate and extreme capacity distributions under high and low churn.

	High Churn					
Target Split	1	2	4	1	20	50
Idspace Split	1	1.68	2.99	1	6.14	5.89
Msg Load(B/s)	711.30	765.04	853.78	736.63	922.48	923.40
Relative Selns	1	1.78	3.13	1	5.74	5.29
Seln p-values	0.000	0.231	0.054	0.000	0.000	0.002
	Low Churn					
Target Split	1	2	4	1	20	50
Idspace Split	1	1.95	3.98	1	23.59	37.41
Msg Load(B/s)	261.73	290.59	318.45	297.01	474.06	413.49
Relative Selns	1	1.98	4.02	1	23.23	34.44
Seln p-values	0.000	0.645	0.774	0.000	NaN	0.001

classes 2 and 4 do match the ideal curve closely enough that they succeed the p-value test, but during high-churn, nodes in the higher capacity classes are still less likely to get selected than they should ideally be. [3]

Figure 4.5 shows why KRB underperforms under high churn: The system-wide objective function (Equation 4.3, Section 4.3) settles to a more or less stable positive value in the presence of the steady churn. KRB's attempts to improve the objective function value below this stable value using node movements are exactly counterbalanced by the effects of node churn, indicating that this is the best KRB can do under this high churn. Increasing the frequency of KRB node movements here does not lead to an improvement in performance, as becomes

[3]The ideal selection curves for Swaplinks and KRB (from Figures 4.2 and 4.4) differ from each other because the number of successful selections performed differ in the two cases.

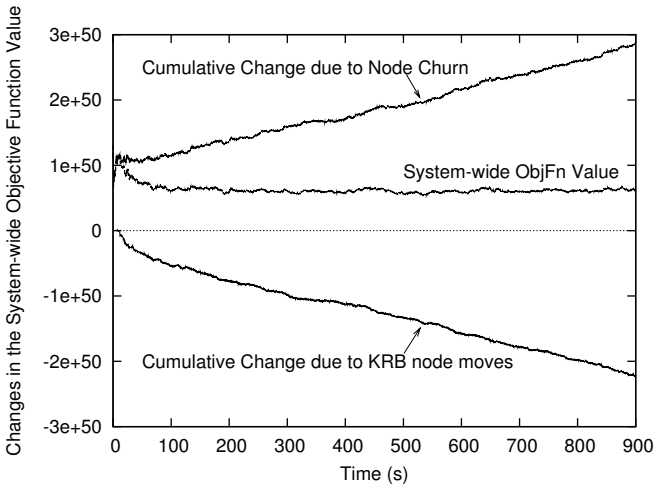


Figure 4.5: Change in the universal objective function as a result of KRB node moves and node churn in a high-churn, moderate-capacity simulation.

clear next.

We evaluated the relative ID-space distribution realized by KRB under high churn and moderate capacities for various values of the KRB parameters (ping, alarm, KRB periods, KRB-threshold), and we found that the combination of the parameters we present here results in the best ID-space distribution. In general, we found that more frequent pings and alarm messages of Bamboo resulted in better results (as can be expected), while there generally was an 'optimal' KRB message frequency and an optimal value for the KRB threshold given the frequencies used for the other messages. Setting the KRB message frequency to higher values resulted in an increase of the number of incorrect KRB moves, where nodes switched positions based on an incorrectly perceived local state, thereby worsening the ID-space distribution. Among the combinations of parameters we tested, the worst performing set yielded about 50% less accurate selection than the setting we use. This experience indicates that it is harder with

KRB to decide on the exact set of various parameters to use in a general setting.[4]

KRB achieves a higher average message load (across all nodes) than does Swaplinks: this is mainly a result of the increased message rates we used to improve KRB's capacity-based ID space distribution. We however do not think that the message load values are high enough to be a concern here.

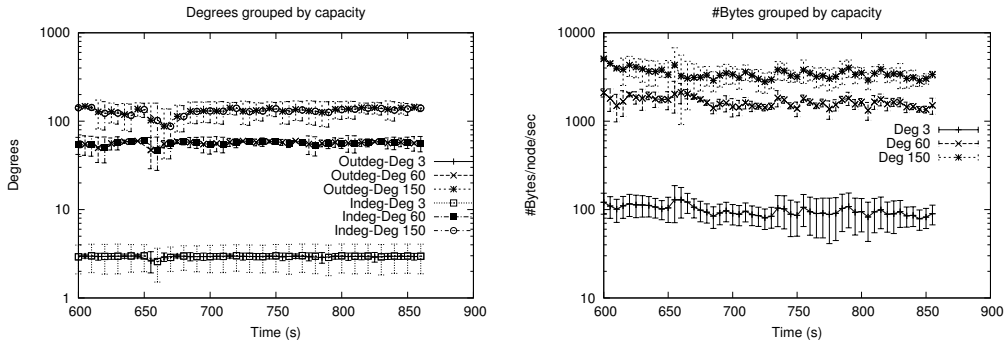
4.4.2 Extreme churn

We now look at the reaction of Swaplinks and KRB to more extreme churn events. The first such event is the entry of a flash-crowd that leads to the network size doubling from 1000 nodes to 2000 nodes in a span of 10 seconds. The second is a "mass departure", where a half of the system population dies instantaneously.

Figure 4.6 shows the results of the Swaplinks flash-crowd scenario under a 3:60:150 degree distribution under high-churn (a median node session time of 2 minutes). The flash-crowd appears in the period 650-660 seconds after the system is started, and two sets of burst-selections are performed starting at 723 seconds and spanning 100 seconds. Table 4.4 summarizes the flash-crowd results over the last 175 seconds of the experiment for both the moderate and extreme capacity distributions.

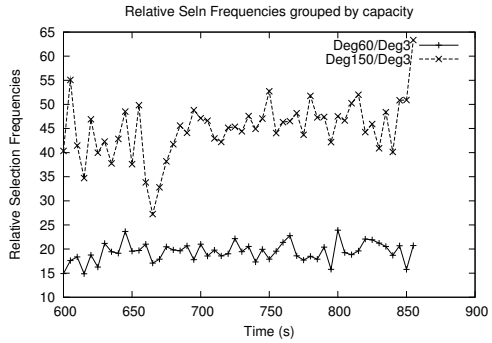
Figure 4.6 shows that while there is a temporary deterioration in all the metrics of interest for a short duration of time immediately after the entry of the

[4]In the search for the best combination of KRB parameters, we did not try out sub-second values for the different parameters: We could conceivably use sub-second values, and achieve better results, but we did not consider this option due to the enormous amount of load it places on the network.



(a) Degree Distribution

(b) Message Loads



(c) Relative Selection Frequencies

Figure 4.6: Swaplinks: Flash-crowd with high churn and extreme capacity distribution

flash-crowd, the system quickly recovers to re-establish desired behavior. The Swaplinks graph in fact generally benefits from nodes entering the system, since this pushes the average degree distribution across the graph towards the ideal value; a comparison of Table 4.4 with Table 4.1 shows that the average values for the degree and relative selection frequencies in fact improve as a result of the arrival of the flash-crowd!

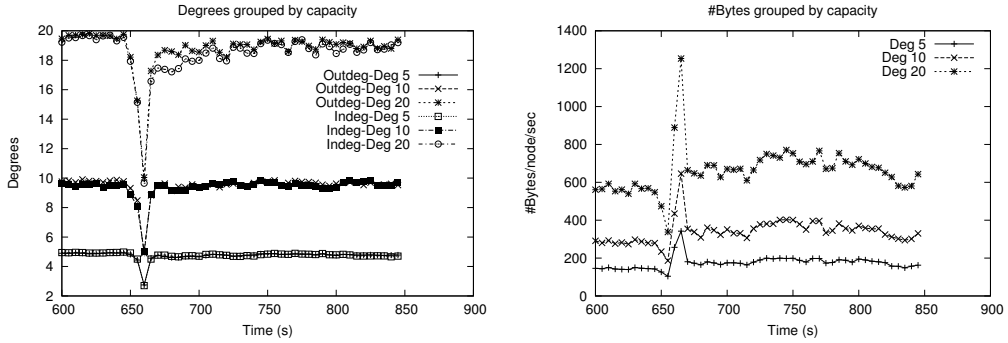
Figure 4.7 and Table 4.4 show results of the Swaplinks mass departure experiments. The mass departures occur at 649 seconds after system start, and burst selections are performed at 719 seconds after system start. From figure 4.7 (for high churn and moderate capacity distributions), we observe that the network suffers for a short duration of time immediately after the huge perturbation, but

Table 4.4: Swaplinks performance with flash-crowds and mass departures under high churn.

	Flash Crowd					
Target Outdeg	5	10	20	3	60	150
Avg Load(B/s)	146.88	291.94	578.49	91.29	1553.79	3320.49
Relative Load	1	1.98	3.93	1	17.01	36.34
Avg Totaldeg	9.82	19.73	39.18	5.9	114.61	269.54
Relative Selns	1	2.06	4.02	1	19.66	46.94
Seln p-values	0.936	0.935	0.722	0.751	0.873	0.567
	Mass Departures					
Target Outdeg	5	10	20	3	60	150
Avg Load(B/s)	179.52	351.01	681.62	121.57	1886.01	4343.61
Relative Load	1	1.95	3.79	1	15.5	35.66
Avg Totaldeg	9.54	19.11	37.59	5.73	102.19	251.1
Relative Selns	1	2.04	3.96	1	17.87	45.84
Seln p-values	0.457	0.912	0.988	0.338	0.418	NaN

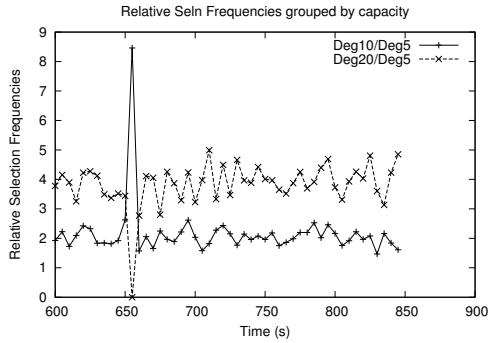
things start to improve thereafter. The message loads and the selection frequencies recover to re-approach the desired 1:2:4 split of message loads and selection frequencies. The extreme capacity results from Table 4.4 also look encouraging: the degrees and the relative selection frequencies are similar to the high (stable) churn, extreme-capacity results shown earlier in Table 4.1. Overall, these experiments demonstrate that Swaplinks is robust to various kinds of network churn under widely different capacity distributions, and that it manages to retain its fine-grained sensitivity to the desired heterogeneity under these conditions.

Table 4.5 summarizes KRB results from the last 175 seconds of the flash-



(a) Degree Distribution

(b) Message Loads



(c) Relative Selection Frequencies

Figure 4.7: Swaplinks: Mass departures with high churn and moderate capacity distribution

crowd and the mass departure simulations for only the moderate capacity distribution under high churn. The flash-crowds and mass departures occur at the same times as those reported in the Swaplinks experiments. The results indicate that the KRB performance suffers significantly as a result of the extreme churn induced. The relative ID-spaces and selection frequencies differ markedly from the target values, resulting in a failure to realize the desired selection distribution. We noticed that while KRB had started to recover from the flash-crowd to approach its stable ID-space distribution towards the end of the simulation, in the mass departure simulation its stable ID-space distribution deteriorated after the mass departures, leading to worse relative selection values at the end of the

Table 4.5: KRB Results for flash-crowds and mass departures for moderate capacity distributions and high churn

	Flash-crowd			Mass Departures		
Target Split	1	2	4	1	2	4
Idspace Loads	1	1.52	2.41	1	1.19	1.68
Relative Selns	1	1.55	2.47	1	1.14	1.75
Seln p-values	0.00	0.00	0.01	0.00	0.48	0.00

simulation. [5] Since we have already seen that KRB fails to adapt to the extreme-capacity setting under high churn, we do not subject it to the more demanding circumstances of both extreme churn (mass departures and flash crowds) and extreme heterogeneity.

4.4.3 Evaluation over PlanetLab

We evaluated Swaplinks over PlanetLab by deploying a 250-node network over 50 PlanetLab hosts distributed across the world. We scaled down the number of selections performed in the burst mode here to about 2500.

Figure 4.8 shows the variation of average node degrees, message loads and the relative selection frequencies with time in a high-churn moderate capacity experiment , and Table 4.6 summarizes both the high-churn and low-churn experiments. While the node-degree curve in the high churn case is not completely stable, due to the high churn, all the values nevertheless adhere rea-

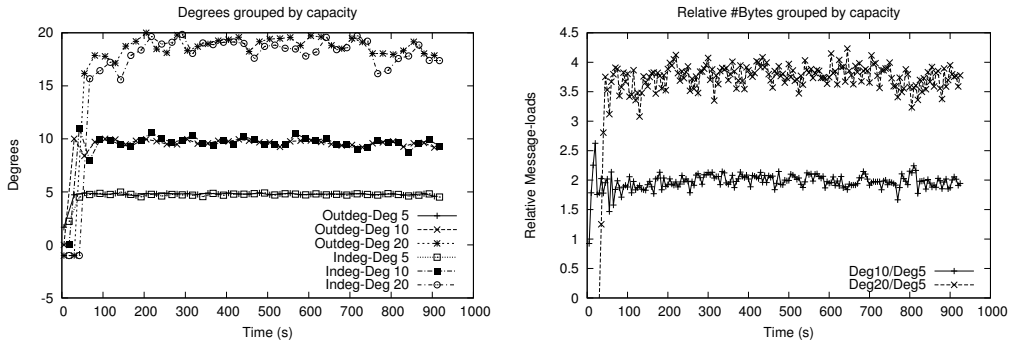
[5]The single positive p-value result here seems to be a lucky one for the nodes in the second capacity class – the smallest capacity nodes get more of the selections than their fair share while the largest get fewer, leaving the capacity 2 nodes with the number of selections closest to its fair share (while still less than it)

Table 4.6: PlanetLab results with moderate capacity distribution

	High Churn			Low Churn		
Target Outdeg	5	10	20	5	10	20
Avg Load(B/s)	210.88	418.36	794.16	196.81	384.10	745.75
Load split	1	1.97	3.76	1	1.95	3.76
TotalDeg	9.54	19.15	37.46	10.04	19.79	39.48
Relative Selns	1	2.07	3.99	1	2.01	3.94
Seln p-values	0.000	0.000	0.001	0.000	0.023	0.002

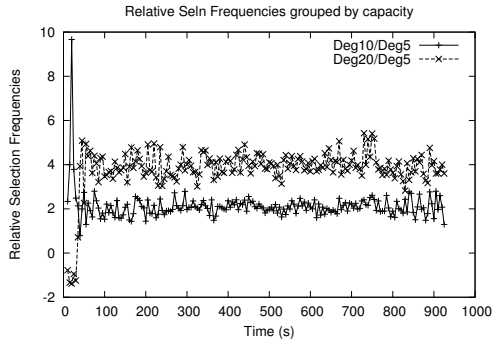
sonably closely to the desired 5:10:20 ratio. But there is a gap between the ideal distribution of #Nodes vs Selection Frequencies and the actual distribution here, leading to poor p-values for the selection distribution. We observed that a few of the planetlab nodes hosting our experiments appeared to freeze occasionally, causing the Swaplinks instances hosted on these nodes to be eventually excluded from the neighbor-sets of other Swaplinks instances. This also means that such nodes would not be selected by any subsequently launched random selection walk, thus causing the discrepancy between the actual and observed selection distributions. In effect, the above freezing behavior contributes an amount of churn not accounted for in our p-value computation, leading to the apparently poor p-values. Note however that the relative selection numbers do adhere quite closely to the desired split.

We do not show results for the extreme capacity distribution here: the fact that each high capacity class constitutes just 1% of the total node population means that there would be too few high capacity nodes in a 250-node experiment to draw reliable conclusions.



(a) Degree Distribution

(b) Message Loads



(c) Relative Selection Frequencies

Figure 4.8: PlanetLab 250 nodes with high churn and moderate capacity distribution

4.4.4 Smart-Pinging

The bulk of the message load seen by Swaplinks nodes is from the heart-beat messages used to determine when a neighbor is down. We would like to minimize this load, in part because in extreme heterogeneity situations some nodes have many neighbors, and in part because a given application might result in a computer belonging to many P2P networks, and therefore having many neighbors. Our basic approach to minimizing heart-beats is as follows: Rather than have every neighbor determine for itself whether a node A is down, one neighbor (at a time) determines if a node A is down. If a neighbor determines that node A is down, it informs the other neighbors of node A, using a flood, that

node A is down.

Specifically, the *smart-pinging* scheme we designed works as follows: Node A tells each of its neighbors about some random set of its other neighbors, such that each neighbor is known by at least some small number of other neighbors. Node A sends each neighbor in turn a small series of (say five) heart-beat messages, each spread two seconds apart. For example, node A sends five heartbeats to neighbor 1, followed by five heartbeats to neighbor 2, and so on. Each neighbor knows when to expect its series of heartbeats, based on timing information conveyed during the previous series of heartbeats. If a neighbor misses all of its heartbeats, it informs all the neighbors of A it knows of that node A is down. These neighbors in turn inform the neighbors they know, and the ensuing flood of packets quickly informs all neighbors that node A is down.[6]

Smart-pinging reduces the amount of bandwidth consumed under no churn, at the cost of a burst of messages that occurs when there is churn, and the possibility of incorrect notifications of node departure. While we need to explore these trade-offs in greater detail, we have currently implemented a preliminary version of smart-pinging. In the current implementation, if node A has d out-neighbors, A has each of its neighbors know of $2\ log_2(d)$ of its (A's) neighbors. Table 4.7 summarizes the results over the second half of the duration of the experiment. This experiment was run with just 8 periodic selectors (instead of 80 as in the previous cases), to isolate the heart-beat load. We observe that smart-pinging does indeed result in a saving on message load at high-capacity nodes under low-churn scenarios.

[6]Structella [11] uses a similar mechanism to reduce heart-beat loads in maintaining leaf-sets, but their mechanism is not applicable in maintaining any arbitrary set of neighbors.

Table 4.7: Smart Pinging: moderate capacity, low churn

Target Outdeg	5	10	20
Avg Load (B/s)	38.23	57.97	89.21
Relative Load	1	1.50	2.30
Avg Totaldeg	9.99	19.90	40.01
Relative Selns	1	2.06	4.08
Seln p-values	0.831	0.904	0.877

4.5 Conclusions and Discussion

Node heterogeneity, where different nodes have different capacities, is an important issue in current peer-to-peer systems. In this chapter, we provide the implementation and performance evaluation of the Swaplinks heterogeneous graph construction and peer selection mechanism. We also compare its heterogeneous selection properties with that of *KRB*, a structured P2P approach derived by adapting the Karger-Ruhl load-balancing scheme to node ID spaces in the Bamboo DHT.

We find that while Swaplinks generally gives good performance along all metrics of interest, KRB finds it hard, under relatively high churn rates, to maintain the desired selection probabilities even for moderate distributions in desired selection probabilities. Also, with KRB, it is non-trivial to zero in on a good set of tuning parameters to use in a general setting. Overall, we find that Swaplinks outperforms KRB in performing heterogeneity-sensitive random peer selection.

While we tested only the one DHT-based approach in this study (mainly be-

cause most other related DHT-based approaches were not suitable in our context), we believe that our finding above could be generalized to a broader statement about the relative merits of unstructured approaches over structured approaches in solving the problem of heterogeneous selection. Heterogeneous selection is basically an unstructured problem – the only requirement on the found nodes is that they be chosen at random, with higher-capacity nodes given proportionately greater preference. The relatively simple unstructured approaches are capable of solving this problem well, while the richer functionalities offered by structured approaches are not required in this setting.

As mentioned in the previous chapter, a limitation of Swaplinks is that it has no defense against misbehaving nodes. For instance, if a node wished to obtain a huge number of neighbors (for instance to DoS a file-sharing application), Swaplinks has no mechanism to prevent this. While we are interested in exploring such mechanisms, Swaplinks is currently only appropriate for use with trusted P2P software. In terms of enhancements to Swaplinks, we need to experiment further with smart-pinging, for instance to insure that it doesn't suffer from false negatives.

We have exercised Swaplinks by using it as a basis for a number of P2P applications, like the Chunkyspread P2P multicast system [104], an extension of the STUNT toolkit for NAT traversal in P2P applications [37], and an experimental P2P file backup system.

4.6 Acknowledgments

We would like to thank Grant Goodale and Victoria Krafft for help in adapting the Bamboo simulator to KRB, Sean Rhea for making the Bamboo simulator available in the first place, and Vidhyashankar Venkataraman for the transit-stub code used in the evaluations.

CHAPTER 5

FINDING THE NEAREST PEER IN P2P NETWORKS

5.1 Introduction

In many peer-to-peer applications, it is beneficial for communicating peers to be close to each other. For example, in online games with direct interaction between gamers, user perceived experience closely depends on the latency between the interacting hosts. In first person shooter (FPS) games, for instance, an increase of latency from 20 to 40 milliseconds noticeably degrades user-perceived performance [77]. Many P2P games in fact only work with the high bandwidths and low latencies seen over LANs, resulting for instance in websites devoted to organizing LAN Parties (e.g., lanpartymap.com). In P2P file-sharing applications, file downloads are faster and more efficient when peers are close to one another: downloads between peers on the same campus network may be orders of magnitude faster than between even nearby peers over the general Internet.

The problem of discovering the closest peers in terms of latency has been an active area of research in the recent past, and a number of solutions have been proposed. Example scalable approaches include: (i) Distance-based sampling, where each peer places other peers it knows into rings or balls of varying sizes, with closer peers tracked more often than those farther away [48, 111], (ii) Solutions based on network-coordinates, where each peer is given a coordinate indicating its "position" in the system, such that the latency between any two peers can be approximated by a function of their coordinates [108, 17]. (iii) Identifier-based sampling, where each peer has an identifier, and tracks other peers with

identifier-prefixes matching its own [42, 13].

All of these approaches use the measured inter-peer latencies to drive their operation. In spite of the disparity of the approaches, they all share the following mechanism: A search for the closest peer to a given peer starts off from a random peer (or a set of random peers), selects among the neighbors of those peers to find closer peers, recursing until it discovers (ideally) the desired closest peer.

For this search process to work scalably and efficiently, the following condition must hold: When a peer P_1 is handling the search for the nearest peer of peer P_2, P_1 should be able to efficiently find a closer peer to P_2 if one exists. In this chapter, we argue that while this condition may hold as long as all peers are relatively far apart, *they do not always hold at all points of the search when peers are very close to each other*. Specifically, the search may not ultimately discover the closest peer if the closest peer happens to be on the same campus network or extended LAN, and therefore the distance to such a peer is measured in microseconds, not milliseconds.

The problem in this case arises out of the way the "last-hop" Internet is laid out. Each ISP has some number of PoPs (Points of Presence) that are used to provide Internet access to its customers. Typically, for a given host to send a packet to any other host not in the same local (campus or LAN) network, the packet must first travel to the given host's PoP. This is often true even if the two hosts share the same PoP and are geographically near each other. Essentially, the last-hop topology resembles a *star-network*, with the PoP as the star node.

As a result, all hosts that gain access through the same PoP and that are

at about the same latency from the PoP end up also being about the same latency from one another. This detail makes it hard to distinguish between the different peers connected to a PoP by looking at the inter-peer latencies alone; various assumptions made by the different closest-peer algorithms, like the *growth-constrained assumption*, *doubling assumption*, and *low dimensionality* (discussed later) all fail to hold around the peers connected to a PoP. This transforms the search for the nearest peer into a brute-force probing of the peers connected to the PoP, making it hard to scalably discover the one other peer in the same campus network from all the different peers connected to the same PoP.

An inability to find the nearest peer represents a significant "opportunity cost": Peers that share the same extended LAN have latencies an order of magnitude smaller, and bandwidths an order of magnitude larger, than those in different networks. The ability to discover peers in the same extended LAN therefore translates to a similar order of magnitude improvement in performance of the application (e.g., gaming, P2P streaming, file-sharing), and in many cases may make the difference between being able to run a given application at all. Also, among applications like P2P streaming and file-sharing, significant savings in bandwidth costs are achieved if bulk data transmission happens between peers in the same network, rather than across the network boundary.

The focus of this chapter, then, is to try to better understand this phenomenon and its implications for proximity systems. In Section 5.2, we provide a detailed explanation of how the large number of relatively equidistant peers served by a PoP poses a problem for closest-peer finding algorithms. We then present large-scale latency measurements over DNS-servers and real (Azureus) P2P end-hosts to indicate that this condition does indeed happen

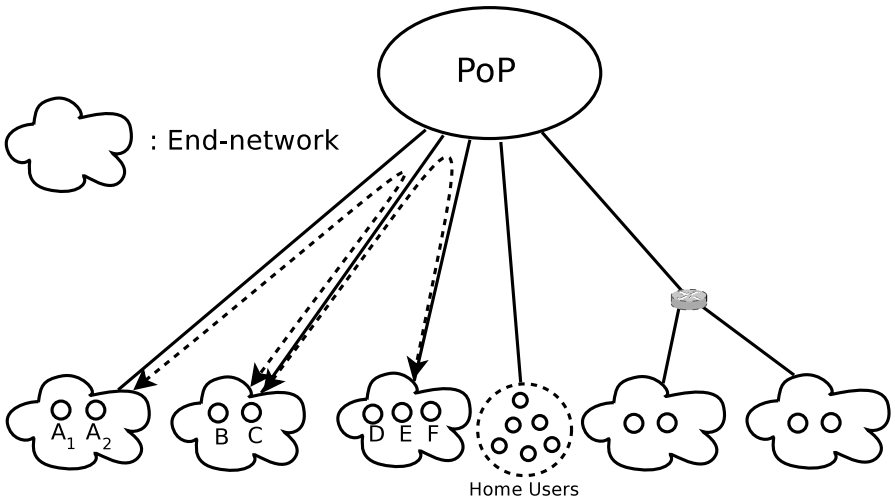


Figure 5.1: Typical connections from a PoP.

to a non-negligible extent in real scenarios (Section 5.3). We next use simulations of Meridian [111], a successful closest-node algorithm, to demonstrate the difficulty caused by the condition in finding the nearest peer (Section 5.4). In Section 5.5, we suggest different possible approaches to tackle this issue: these approaches explicitly or implicitly search for peers that are *topologically* close to them. We conduct a brief evaluation of two of these approaches and show that one of them is very likely to succeed in real settings. We conclude in Section 5.6.

5.2 The Last-Hop Clustering Effect in the Internet

The Internet "last hop" provides access to end-hosts: ISPs deploy PoPs at well-populated areas, and run physical connections from end-hosts or networks of end-hosts to routers in nearby PoPs.

Figure 5.1 shows a typical graph of connections from the PoP. We use the

term *end-network* to denote a network of end-hosts all in the same geographic location, e.g., LANs, extended LANs, and campus and corporate networks. An end-host's *local-network* is the end-network that it resides in. It is possible that end-hosts are not part of an end-network; these would typically be hosts at homes (broadband / DSL / dial-up users). Looking at Figure 5.1, connections funnel in from the end-hosts and end-networks, possibly merging as they get closer to the PoP.

Suppose now that a message is sent from one of the end-hosts served by the PoP to another. We assume here that if the path from the message-source to the PoP and the path from the message-destination to the PoP share a closer upstream router than the PoP, then the message would only need to go up to the common router and then down to the destination. If both the source and the destination are in the same end-network, we assume that the message would be routed entirely within the end-network and that the corresponding latency would be much smaller than if the message had to traverse different end-networks. If the paths do not share a closer router than the PoP, and the source and destination hosts are in different end-networks, then the message needs to go all the way up to the PoP and then back to the destination. Measurements in Section 5.3.1 validate these assumptions.

In the following text, we restrict our attention to those end-networks where messages sent from a host in one end-network to a host in another end-network need to traverse the PoP, i.e., end-networks whose paths to the PoP share no common routers. Within this set of end-networks, we only consider those end-networks that are at about the same latency from the PoP – for our purposes, we consider two end-networks to be at about the same latency from the PoP if

the latencies are close enough to each other that nearest-peer algorithms cannot reliably distinguish peers based on the difference between these latencies. We call the set of hosts in this set of end-networks the PoP's *cluster*. Now, if the only way to distinguish between different hosts is based on the latencies to the hosts, the above construction results in the following properties: (i) Any two hosts inside the cluster appear indistinguishable to any host outside the cluster, (ii) Similarly, any two hosts, say A_1 and B, which are inside the cluster, appear indistinguishable to any host C that is also inside the cluster, but outside the end-networks of A_1 and B (see Figure 5.1).

Now say there is a P2P network that consists of a few hosts from each of the end-networks in the cluster, and that each newly joining peer wants to find its closest peer. We assume that the closest-peer algorithm used here initiates a closest-peer *query* at a random peer when a new peer enters the system. In line with previously proposed solutions, we assume that the peer currently handling the query selectively probes other peers it knows in order to find a peer that is closer to the new peer. This is repeated until the closest peer is found. We assume here that the only information about a peer that the algorithm uses is its latencies to other peers (or non-peer nodes), again in line with previous solutions to this problem.

Assume now that peer A_1 has already joined the P2P system, and that peer A_2 now enters the system (see Figure 5.1). The closest-peer query for A_2 starts off from a random host, progressively finding peers closer to A_2, and might eventually reach one of the peers (say C) inside the cluster. Because C is virtually a randomly picked node from the entire cluster, it is likely to be not in A_1's local network. Ideally, the closest-peer query would eventually reach A_1,

finding it (A_1) as A_2's closest peer. But from C's point of view, all peers in the cluster (other than those in C's local network) appear identical to one another: For instance, C cannot tell which of peers A_1, D, and E, all inside the cluster, is closer to A_2. Measurements to nodes outside the cluster are of no use here, since all peers inside the cluster appear to be at the same latency from any node outside the cluster. So the best C can do now is to hand off the query to some other peer in the cluster, in the hope that the other peer is closer to the target A_2. The same holds true for all the peers that handle the query from this point on: The only "intelligence" each of these peers can employ in choosing the next peer is to forward the query to a peer not in its own local cluster. Thus we conclude that the query, if it does eventually reach A_1, will have traversed through, on average, a number of peers equal to the number of end-networks in the cluster before it gets there. This translates to a lower bound on the number of latency "probes" performed as well: Since A_2 is a new peer entering the network, for a peer to tell if it is the closest peer to A_2, it has to first measure its latency to A_2. [1]

In effect, there is a phase-transition in the performance of the algorithm once the query enters the cluster. Prior to entering the cluster, the algorithm might have made rapid progress in finding closer and closer peers, but once it enters the cluster, it is stuck trying to probe peers in the different end-networks in a brute-force manner. This means that when the number of end-networks in the cluster is large, finding the closest peer in the same end-network might be infeasible, since it requires a brute-force search through the different end-networks.

[1] The one exception to this is coordinate-systems, which do not need explicit latency probes for each new latency estimate. We discuss coordinate-systems in Section 5.2.2.

5.2.1 The Clustering Condition

The above line of reasoning is unchanged if we replace the PoP by any set of nearby routers (with negligible latencies between one another). This is important from the point of view of measurement, since accurately identifying a PoP is a hard problem. So the requirements for a cluster as described above are as follows:

1. The cluster is made of a large number of peers in different end-networks.
2. Any message sent over the Internet from a peer in one end-network of the cluster to a peer in another end-network of the cluster passes through at least one router that is part of the *cluster-hub*, a set of close-by routers.
3. All end-networks in the cluster are at about the same latency from the cluster-hub. Again, by "about the same latency", we mean that the latencies are close enough that the nearest-peer algorithm being used cannot reliably use the differences in these latencies to tell apart the different peers. How close they need to be depends on the particular algorithm being used.

We denote this the *clustering condition*. If a set of peers satisfies the clustering condition, it will be hard to find the closest peers to peers in the cluster. Measurements presented later in the chapter, in Section 5.3, indicates that the clustering condition does occur in real settings with non-negligible probability.

5.2.2 Common Assumptions Behind Nearest-Peer Algorithms

Many of the previously proposed nearest-peer algorithms make assumptions about the inter-peer latency distribution, allowing them to need a provably small number of latency measurements. We now examine a few such commonly used assumptions, and illustrate how they fail to hold under the clustering condition.

Growth Constrained Metrics: The space in which the peers are located is said to be *growth-constrained* if the following condition holds: Given any peer P, and latency l, the number of all the peers within latency $2l$ from P is not significantly larger than the number of all the peers within latency l from P [111, 42]. When the nearest peer to peer P is to be found in a growth-constrained metric, one can start from a random peer and zero in on the closest peer by repeatedly probing neighbors and progressively finding closer peers. Progress is ensured by the growth-constrained assumption, as at any point in the search, each peer is assured of having enough neighboring peers that are closer to the target peer than itself. Plaxton et al [76], Karger and Ruhl [48], and Tapestry [42] give nearest-peer algorithms that make the growth-constrained assumption or close variants of the assumption.

Under the clustering condition, however, the space around the cluster does not conform to the growth-constrained assumption: Given a peer P inside an end-network in the cluster, we see that there is a small number of peers at very small latencies from P, and an empty space not occupied by any peers for a significant distance. This is immediately followed by a well-populated region containing other peers in the cluster. If the other peers in the cluster are between latencies l and $l + \delta$ from peer P, and $\delta \leq l$, the number of peers that are within

latency $2l$ from P is significantly larger than those within latency l, thus violating the growth-constrained assumption.

Doubling Assumption: A set of peers is said to be covered by a *ball* of radius r if the latency between any two peers in the set is less than or equal to $2r$. Under the doubling assumption, any set of peers covered by a ball of radius r can be covered by a small number of balls of radius $\frac{r}{2}$. The doubling assumption is more general than the growth-constrained assumption: A space that satisfies the growth-constrained assumption also satisfies the doubling assumption [96]. The doubling assumption suggests the following approach to find the nearest peer: Say peer B wants to find the nearest peer to peer A. If both A and B are covered by a ball $ball_{AB}$, there should be a small number of smaller balls that cover the set covered by $ball_{AB}$. If B now can find some peer inside the smaller ball that covers A, progress is achieved. The Meridian closest node algorithm [111] makes the doubling assumption.

The doubling assumption also fails under the clustering condition. Consider the smallest ball that covers all the peers in a cluster: The radius of this ball is the same as the latency of the different peers to the common upstream router(s). Any ball of half this radius would cover only those peers in a single end-network. Thus the number of smaller balls required to cover the larger ball is on the order of the large number of end-networks in the cluster, thereby violating the doubling assumption.

Low Dimensionality: Under this assumption, the latency-space can be embedded with very little error into a low-dimension space, usually Euclidean. Peers then have coordinates assigned to them, and latencies between any two peers can be estimated using the coordinates without having to resort to ac-

tive measurements between the two peers. Example approaches here include Mithos [108] and PIC [17]. However, where the clustering condition holds, the latency-space around the cluster has high dimensions: the number of dimensions is on the order of the number of end-networks in the cluster.

5.2.3 Behavior of Sample Nearest-Peer Algorithms Under the Clustering Condition

We now examine how a few specific nearest-peer finding algorithms would fare under the clustering condition: Meridian [111] is an algorithm designed to find the closest node from among several nodes (e.g., servers) to a given target (e.g., a client that wants to find the closest server from a set of servers). Meridian builds an overlay of the participant nodes, with each node organizing other nodes into rings of different radii: Other nodes close to a given node will occupy the nearer rings of the given node, and vice-versa. Each ring can have up to a maximum number of nodes. Members of a ring are also chosen so that they have a high hypervolume. In order to find the closest node to a given target, Meridian initiates a query starting from a random node. The node currently processing the query measures its latency to the target, and asks the nodes in its rings that it knows are at about the same latency to itself to measure their latencies to the target. The query is then forwarded to the node with the minimum distance to the target. The query terminates at any step when no significant reduction is achieved in the latency to the target from the closest node discovered so far. [2] When the underlying latency space satisfies the doubling assumption, the fact that the members of a ring have high hypervolume and thus are far apart from

[2]The Meridian parameter β specifies what the minimum reduction must be.

one another helps Meridian efficiently pick a closer node to the target.

To use Meridian to find the closest peer to a given *peer*, we would just have to run a Meridian query with the given peer as the target. Under the clustering conditions described above, the query would eventually reach one of the peers, say peer P in the cluster. Since almost all of the other peers (barring those in P's end-network) are at the same latency to P as P is to the target peer, the set of peers next asked to measure their latencies to the target is practically just a randomly chosen set from the entire cluster. The hypervolume maximization does not help here, since this space does not satisfy the doubling assumption: Any set of randomly chosen peers from the cluster has about the same hypervolume, so almost all peers in the cluster would be equally good (or bad) choices as ring members. Thus the only way the query would reach the correct end-network is by random chance. Accordingly, the query will terminate quickly, and likely not in the same end-network as the target.

The PIC [17] algorithm assigns each peer a multi-dimensional Euclidean coordinate that approximates its "position" in the latency-space. In order for a peer to find its closest peer, it first computes its (rough) coordinates, and then launches multiple greedy walks aimed at finding closer peers: At each hop of the walk, the walk chooses the closest neighbor as predicted by the respective coordinates as the next hop [3]. However, under the clustering condition, to assign coordinates to each peer without error would need an impractically huge number of dimensions. With a small number of dimensions, all peers within a cluster would end up having almost the same coordinates, thus making it impossible to tell them apart, and ensuring the search for the nearest peer most

[3]PIC also has a variant where the new peer's coordinates are repeatedly recomputed at each step of the greedy walks, but our argument holds equally well for the variant

likely does not reach the target end-network.

5.3 Clustering Condition in the Internet

In this section, we verify that the clustering condition occurs among real peers in the Internet, and in so doing, validate assumptions made in the previous section. To investigate the existence of the clustering condition, we use a large set of IP addresses of peers in the Azureus P2P network, taken from Ledlie et al's study [58, 69]. We run the traceroute tool from multiple geographically distributed vantage points to identify clusters of peers and their cluster-hubs. To directly check that messages sent from one peer in the cluster to another traverses the cluster-hub, however, we would need to have control over the peers. Since this is not the case, we instead use an alternate measurement setup using recursive DNS servers to show this property. We use the King technique [39] to measure the latency between pairs of DNS servers in a cluster, and compare this latency with the sum of latencies from the respective DNS servers to the cluster-hub. We use experiments over the DNS servers to also verify the important assumption that latencies within end-networks are significantly smaller than latencies across different end-networks.

We present our DNS server latency measurements next, in Section 5.3.1, and then present the clustering results over Azureus peers in Section 5.3.2.

5.3.1 Latency Measurement Results over DNS servers

In this section, we use measurements over DNS servers to address the question of whether messages sent between peers in a cluster traverse the cluster-hub. The basic mechanism we use is as follows: Given a pair of DNS servers in a cluster, we first use the King technique, as described later in the section, to derive an approximate measured latency between the two servers. We then determine the likely cluster-hub in the cluster, and predict the latency between the servers assuming messages between them do pass through the cluster-hub. Finally, we compare the measured latency with the predicted latency: the closer the two are, the stronger the indication that messages traverse the cluster-hub. We use a set of about 22,000 recursive DNS servers, taken from Ballani et al's study [3], as the basis for the measurements.

We use the *rockettrace* utility [97], an extension of traceroute, in the measurement here. In addition to reporting the names and IP addresses of routers on the way to the destination, rockettrace also annotates router names with the router's owning AS (autonomous system) and city where the router is located. We assume that routers annotated with the same AS and city reside in the same ISP PoP. We run rockettrace from a single measurement host to each DNS server, and map each DNS server to its closest upstream PoP on the trace, as given by rockettrace. Thus, for each PoP, we are able to get the cluster of DNS servers that have the PoP as their closest upstream PoP. We then randomly pick pairs of DNS servers from each cluster, such that each DNS server appears in about 4 pairs. We measure the latency between the servers in the pair using the King technique [39]. King first measures the latency from the measurement host (the host that King is being executed on) to one of the recursive name-servers in the

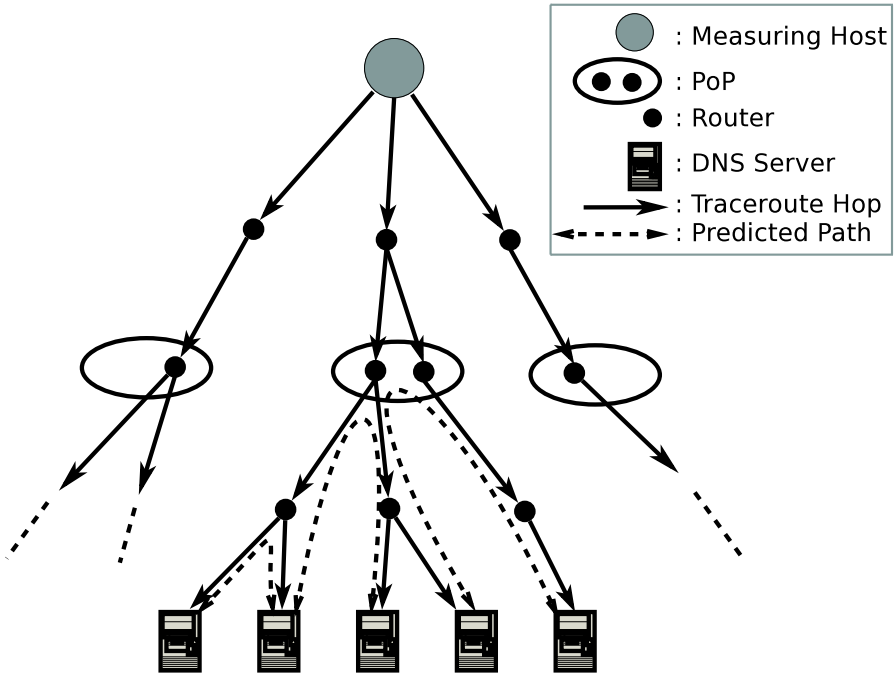


Figure 5.2: A sample tree of traceroutes from the measuring host. For clarity, routes are shown to be smaller than they typically are.

pair. It then sends this recursive name-server a recursive name-query for a name that the second name-server is an authoritative name server for, so the query is forwarded to the second name-server. King is thus able to estimate the latency between the two name-servers.

We predict the latency between two DNS servers in a cluster in the following manner (also see Figure 5.2):

(i) If the rockettrace paths to the two servers share a closer router than the PoP to the servers (i.e., a router that is further downstream to the DNS servers than the PoP), then we predict that messages sent between the two servers travel up until the closest common router, and then back down to the destination. Accordingly, the predicted latency between the two routers is the sum of the latencies from the DNS servers to the common router. We get these latter latencies

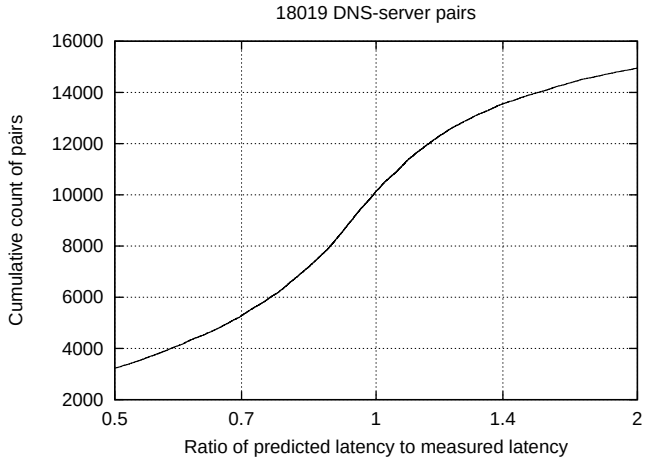


Figure 5.3: Cumulative distribution of the prediction measure.

using the *ping* tool, by subtracting latency to the closest common router from the latencies to the DNS servers.

(ii) If the rockettrace paths share no closer router than the PoP, we predict that the latency between the two servers is the sum of the latencies from the servers to the PoP. The reasoning here is that routers in a PoP are quite close together, and should have negligible latencies between one another. We again measure latencies from the DNS servers to the PoP using the ping tool.

Figure 5.3 shows the cumulative distribution of the *prediction measure*, which we define as the ratio of the predicted latency to the measured latency between a pair of DNS servers. The closer this figure is to 1, the better the accuracy of prediction is. The plot does not include pairs made of DNS servers from the same domain: Such servers are highly likely to be authoritative name-servers for the same names, so the recursive queries used by King may not be forwarded to the second name-server, making King unusable in this scenario. We also discard entries where the computed latencies between a DNS server and the relevant router or PoP turned out to be negative (as a result of the subtraction

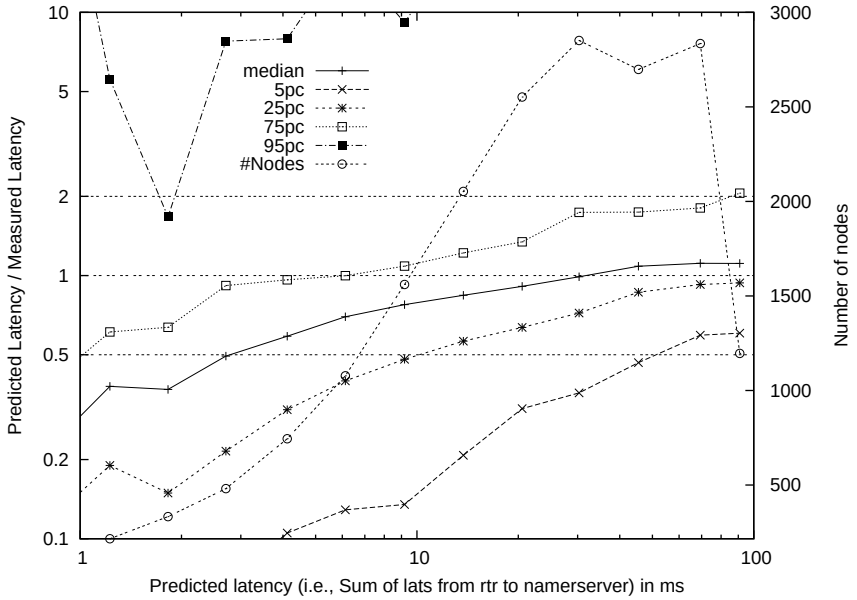


Figure 5.4: Tracking accuracy of prediction as a function of the predicted latency between the pairs.

of the latency to the router from the latency to the DNS server). Finally, we exclude pairs where the DNS servers are more than 10 hops away from their closest common upstream PoP or common router, and pairs where the predicted latency between the DNS servers is more than 100 ms. This is because DNS servers that are farther away will probably have alternate shorter paths between them. After these eliminations, we have a residual set of 18019 DNS server pairs, and Figure 5.3 shows that about 11700 of these, i.e., about 65% of the tested pairs, have prediction measure between the range of 0.5 and 2.

Figure 5.4 shows the prediction measure (median, and percentile values) as a function of the predicted distance between the pairs. The plot is essentially a scatter-plot of the prediction measure versus the predicted latency, but where (for ease of understanding) we group sample points from nearby predicted latencies into a single bin with a representative predicted latency value, and where we display the median and percentiles of the prediction measure for

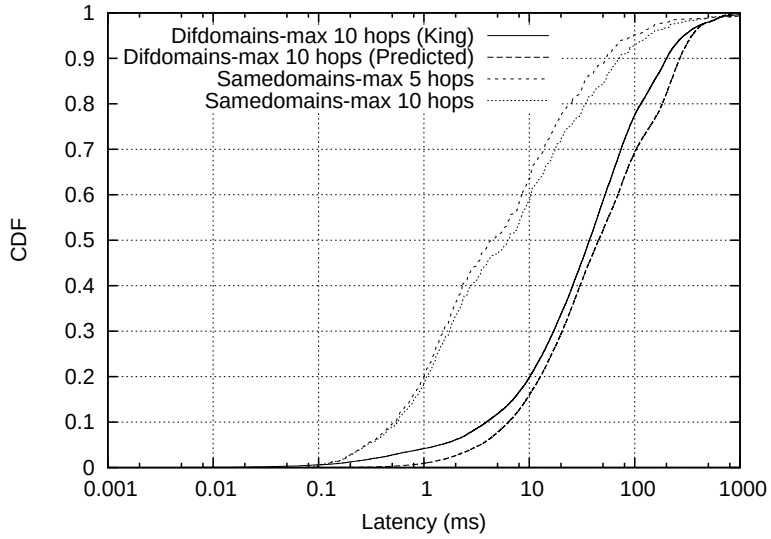


Figure 5.5: Comparing intra-domain latencies with inter-domain latencies.

the sample points that fall in the respective bin.

There is a definite trend visible in the plot that indicates that the prediction measure increases with the predicted latency. In other words, as the predicted latency increases, the measured latency decreases in comparison to the predicted latency. We believe this trend arises due to the following reason: At low latencies, the lag involved at the DNS servers executing the King measurements is likely to constitute a non-negligible part of the measured latency, thus leading to an artificial increase in the measured latency. On the other hand, at large latencies, it gets more likely that there are alternate paths between the DNS servers that do not traverse the common upstream router, thereby decreasing the measured latency and increasing the prediction measure.

In the argument outlined in the previous section, we had assumed that latencies between two nodes in an end-network were significantly smaller than latencies between nodes that are in the same cluster but in different end-networks. We now verify this assumption. To obtain sets of nodes that are more likely to

be in the same end-network than other nodes, we assemble pairs of DNS servers sharing the same domain name. Figure 5.5 compares the intra-domain latency distribution among these pairs with the latency distribution among pairs of DNS servers in different domains. We obtain the two intra-domain curves in the figure by restricting the maximum number of hops between the DNS servers and the closest common upstream PoP or router to, respectively, 5 and 10. We similarly restrict the maximum number of hops in the inter-domain case to 10, since we want to retain only those pairs in the same cluster. We use the predicted latencies to compute the intra-domain latency distribution, since King cannot be used here, as described earlier. We plot both the predicted and King-measured latencies for the inter-domain DNS server pairs.[4]

The figure shows that the intra-domain latencies are indeed much smaller (by about an order of magnitude) than the inter-domain latencies, confirming our assumption. Also, pruning the maximum number of hops from 10 to 5 results in only a modest reduction in the latencies, mainly because very few DNS servers in the intra-domain pairs are farther than 5 hops from their common upstream router. We note here that our method of compiling the intra-domain DNS server pairs is only an approximation of hosts in the same end-network; we noticed cases where the DNS servers in a pair were located in different geographic locations. We therefore expect hosts in the same end-network to have even smaller latencies than those shown in this plot.

A final aspect noticeable from the plot is that the inter-domain predicted latency distribution matches the measured latency distribution reasonably well.

Overall, the results in this section show that latencies between hosts in the

[4]There are about 500 DNS server pairs in the intra-domain distribution, and about 26000 pairs in the inter-domain latency distribution.

same end-network are significantly smaller than that between hosts in different networks. The prediction results, while considerably accurate, are admittedly not as decisive: A non-negligible portion of the predicted distances (35%) lie outside the range of 0.5 to 2. There are factors in addition to those mentioned earlier that possibly lead to errors in the measured data: Firstly, measurements over the Internet are inherently prone to noise, so cannot be expected to give consistently accurate results. Also, rockettrace's method of annotating routers with information about the router's AS and geographical information is based on the name of the router; if the name is mis-configured, this leads to erroneous results. In view of these mitigating factors, we believe that while the results are noisy, they do indicate that most of the nodes in a cluster do need to traverse the closest common router in order to communicate with one another. We extend this finding to peers that are end-hosts (and not servers) as well.

5.3.2 Measurement over Azureus Client IP Addresses

We now examine the occurrence of the clustering property in the Azureus P2P network, using a set of 156,658 Azureus IP addresses collected by Ledlie et al [58, 69]. The basic method is as follows: We track each peer's closest upstream router using traceroutes from multiple vantage points spread across the globe, produce clusters of peers that all have the same upstream router, identify the common upstream router as the cluster-hubs, measure latencies between the cluster-hub and the peers within each cluster, and further prune down the clusters to ensure all cluster peers have similar latencies to the cluster-hub.

The closest upstream router of a peer, as seen from a particular vantage

Table 5.1: The set of Planetlab [75] nodes used as vantage points

Vantage Point	Location
planetlab02.cs.washington.edu	Washington, USA
planetlab3.ucsd.edu	California, USA
planetlab5.cs.cornell.edu	New York, USA
planetlab2.acis.ufl.edu	Florida, USA
neu1.6planetlab.edu.cn	Shenyang, China
planetlab2.iii.u-tokyo.ac.jp	Tokyo, Japan
planetlab2.xeno.cl.cam.ac.uk	Cambridge, England

point, is the last router seen on the trace from the vantage point to the peer. [5] We retain only those peers that have the same upstream router as seen from all the vantage points.

We group peers with the same upstream router into clusters. Table 5.1 shows the set of vantage points used. The fact that these are well-distributed across the globe, and the DNS-server measurement results from Section 5.3.1 indicate that the common upstream router is on the route between any two peers in the cluster. We thus choose the common upstream router as the cluster-hub.

To measure the distribution of latencies from the cluster-hub of the clusters of peers to each of the peers in the cluster, we should be able to first measure the latencies to the peers themselves. But ping and traceroute, the usual tools of choice, mostly fail here: Most peers do not respond to either ping or traceroute with valid latencies. Since the peers here are Azureus clients that communicate over TCP and use a well-known port (6881), we instead measure the latency to

[5] We consider only valid routers here. E.g., if none of the entries in the penultimate hop of a traceroute are valid, we go up to the next hop(s) to get the closest upstream router.

a peer as the time it takes to complete a TCP 'connect' to the port at the peer; we call this the "TCP-ping". Out of the 156,658 total IP addresses in the original list, only 5904 remained that responded to the TCP pings or traceroutes and had a unique upstream router as seen from all the vantage points. We group these peers into clusters and find the latency distribution within each cluster: We launch TCP pings from the same vantage points as seen above to get latencies to the peers. And we use the appropriate entry from the traceroute output as the latency to the cluster-hub, and subtract this from the latencies to the peers to compute the latencies from the cluster-hub to the peers in each cluster.

With the above formation of clusters, it is possible that the hub-to-peer latencies might vary widely within the cluster. So we further pare down the clusters, ensuring that within each cluster, the hub-to-peer latencies are all within a factor of 1.5 from one another. The exact extent of similarity in hub-to-peer latencies that leads the cluster-peers to be indistinguishable in the eyes of a nearest-peer algorithm of course depends on the particular algorithm itself; we use the factor of 1.5 here as an approximation of this.

Figure 5.6 shows the cumulative distribution of cluster sizes, both before and after the pruning step described above. About 16% of the peers are in (pruned) clusters of size 25 or larger. As a sample of the inter-peer latency distribution within clusters, Figure 5.7 shows the distribution of latencies from the cluster-hub to the peers in the cluster for the largest 5 pruned clusters. The latency distribution shown here indicates that peers in the displayed clusters are likely in different end-networks. These results show that even the small sample of 5904 peers has a non-negligible fraction of the population in clusters that satisfy the clustering condition: they have peers spanning reasonably large numbers

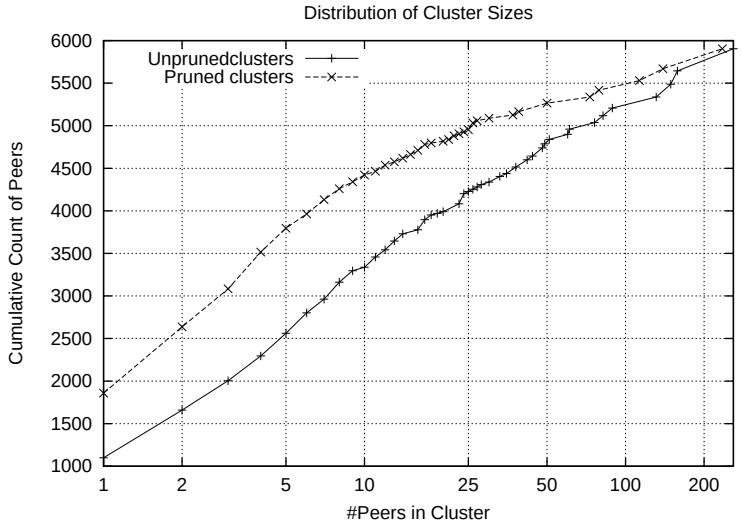


Figure 5.6: Distribution of cluster sizes, both before and after pruning, with 5904 peers in all.

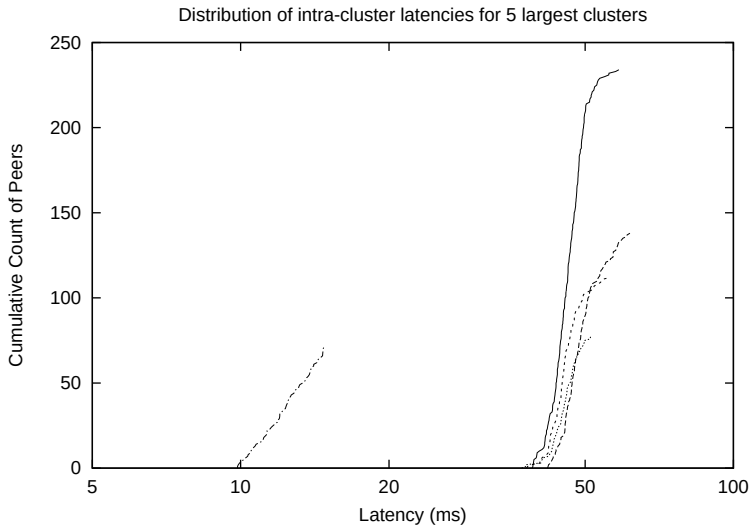


Figure 5.7: Distribution of latencies within clusters for the largest 5 clusters. The cluster-sizes are, respectively, 235, 139, 113, 79, and 73.

of end-networks, and have all peers at similar latencies from one another. Thus new peers sharing end-networks with peers in the cluster will find it hard to discover their closest peers.

We note here that the measurements presented in this section are not (and are not intended to be) an accurate quantitative evaluation of the exact extent of occurrence of the clustering condition – indeed it is almost impossible to do this, without explicit co-operation from the participant peers. Instead, these results should be taken as an indicator that the clustering condition does exist to a non-negligible degree, and that designers of latency-sensitive applications need to keep this in mind.

5.4 Meridian Simulations under the Clustering Condition

Earlier, in Section 5.2, we argued analytically that the different nearest-peer algorithms would find it difficult under the clustering condition to find exact-closest peers. We now use simulations of the Meridian algorithm to help verify this argument. We use the Meridian simulator used in the Meridian paper [111] for the simulations.

To simulate the clustering condition in the inter-peer latency matrix, we create clusters of end-networks that in turn contain peers. Each end-network in a cluster is at a given latency from the cluster-hub of the cluster: the closer these latencies are to one another, the more the cluster conforms to the clustering condition. Within each cluster, we set the mean latency between the cluster-hub and the end-networks in the cluster to be uniformly distributed between 4 ms and 6 ms. We use a parameter δ that quantifies the variation of latencies within a cluster – the latency of each end-network to its cluster-hub is uniformly distributed between $(1 - \delta)$ and $(1 + \delta)$ times the mean latency between the cluster-hub and the end-networks in the cluster. We use the Meridian DNS-server latency

dataset [111, 67] to simulate latencies between the cluster-hubs: each cluster-hub is represented by a randomly picked DNS server from the dataset. DNS-server pairs in the Meridian dataset have a median latency of around 65 ms.

All end-networks in our simulation contain two peers each. Two peers that are both in the same end-network have a latency of 100 μs between them, and identical latencies to all other peers. Two peers in different end-networks have an inter-peer latency equal to the latency between the end-networks that contain them, computed according to the latency assignment in the previous paragraph (where the path starts from one peer, goes up to its cluster-hub, across to the cluster-hub of the second peer, and down to the second peer).

The above assignment satisfies the expected gradation of latencies: latencies within an end-network are more than a magnitude smaller than latencies across end-networks, and latencies within a cluster are smaller than latencies across clusters. We are interested here in identifying recognizable trends that Meridian exhibits with changing clustering properties.

The above setup is used to build inter-peer latency matrices with about 2500 peers, out of which about 2400 randomly picked peers are picked to build a Meridian overlay. The 100 remaining peers are used as *target nodes*, where Meridian tries to find the closest peer in the overlay to chosen target nodes. In each simulation, 5000 Meridian closest-neighbor queries are launched to find the closest peer to randomly chosen target nodes. Note that the target nodes themselves do not join the Meridian overlay, thereby letting reuse of the same target multiple times. Also, since the target nodes are picked randomly from the original set of peers, it is very likely that the target shares the same end-network as some other peer in the overlay, and this peer would be the closest peer in the

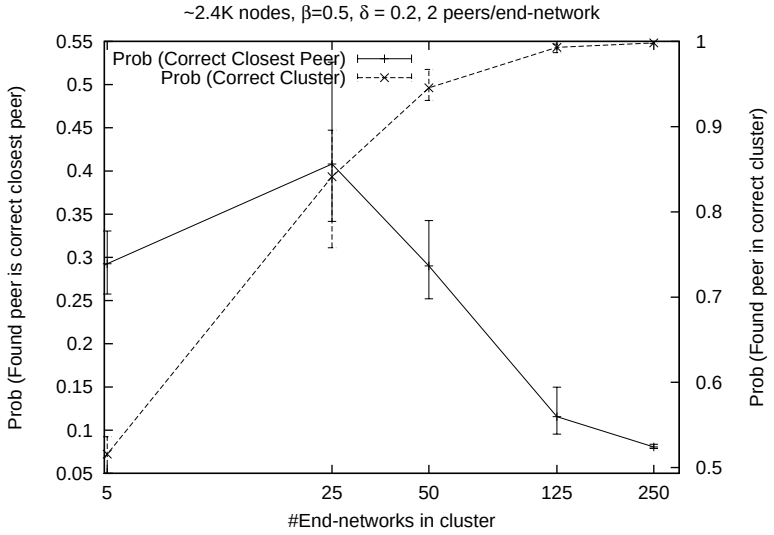


Figure 5.8: Meridian success rates in finding (i) the absolute closest peer, and (ii) some peer in the same cluster as the target node.

overlay to the target.

We ran all of the Meridian simulations with the Meridian parameter β set to 0.5, and the number of neighbors per ring set to 16, as in the Meridian paper. [6] All the numbers presented in this section are the results of three separate simulations, each using a different inter-peer latency dataset.

We first look at the changing performance of Meridian with the change in the number of peers in the cluster. Figure 5.8 shows, as a function of the average number of end-networks in a cluster, the proportion of times Meridian is able to find the correct closest peer and the proportion of times it is able to find a peer in the correct cluster as the closest peer. [7] The correct cluster here is the cluster that contains the target node. We set δ to 0.2 in these simulations. The accuracy of Meridian's choice of closest peer initially improves with an increase

[6] The β parameter in Meridian controls the trade-off between the number of messages sent as part of a Meridian query resolution and the accuracy of the result of the query.

[7] The plotted values are the median, minimum and maximum values across the three simulation runs.

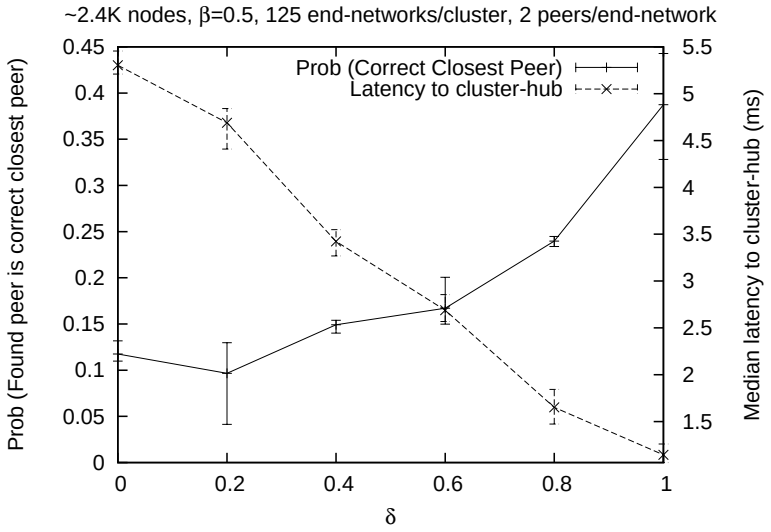


Figure 5.9: Meridian's accuracy in finding the absolute-closest peer and the latency of the discovered peer from the cluster-hub as functions of δ, the variation in latencies within a cluster.

in cluster-size, but falls off at larger sizes, while the probability of finding some peer in the correct cluster uniformly improves with cluster size. The reason for the latter behavior is that with larger cluster sizes, there are more peers from the correct cluster, improving their chance of being discovered by the Meridian queries. At the lower end of the spectrum of cluster-sizes, the accuracy of Meridian's choice of the closest peer also improves with an increase in cluster-size, owing to an increased probability that the query enters the correct cluster in the first place. But beyond a certain point (at x=25 in the plot), the increased likelihood of finding the correct cluster is more than outweighed by the phase transition caused by the emergence of the clustering condition: There is less and less chance that random probing among peers inside the cluster leads the query to the correct end-network. This result shows that the probability of finding the correct closest peer indeed deteriorates when the clustering condition occurs.

We next examine the effect of variations in intra-cluster latencies on the ac-

curacy of Meridian. The parameter δ described above captures this variation [8]. We run Meridian simulations over a range of different values of δ, starting from $\delta = 0$, with no variation in intra-cluster latencies, to $\delta = 1$, where latencies from a peer to its cluster-hub could range anywhere between 0 and twice the average hub-to-peer latency for the cluster. Note here that the larger δ is, the less the network conforms to the clustering condition. We run the simulations with an average 125 end-networks in each cluster. Figure 5.9 shows the results. With an increase in δ, there is a significant improvement in Meridian's accuracy in finding the closest peer. This is a direct result of the clustering condition holding for smaller values of δ, and its weakening at larger values of δ. For larger values of δ, the cluster could effectively be split into smaller clusters, where within each cluster, there is a much smaller variation in the intra-cluster latencies. With smaller clusters, there is a greater likelihood of random probing succeeding, thus leading to better accuracy in finding the nearest peer.

Figure 5.9 also shows the average latency from the cluster-hub to the peer found by Meridian, not counting those cases where Meridian actually found the correct closest peer. The latency decreases with an increase in δ. This is because for higher values of δ, there would be peers that are closer to the cluster-hub (by construction). Peers that are closer to the cluster-hub are also closer to all other peers in the cluster, so Meridian, by design, preferentially picks such peers over others. A side-effect of this is that peers closer to the cluster-hub end up being selected more often than others, increasing the load placed on them. This raises an interesting but hard-to-answer question: Given that it is hard at times to find the closest peer in the same end-network, should we aim

[8] To refresh, the latency of each end-network to its cluster-hub is uniformly distributed between $(1 - \delta)$ and $(1 + \delta)$ times the mean latency between the cluster-hub and the end-networks in the cluster.

to find the closest peer that *can* be found, keeping in mind that doing so would end up overloading a few peers? An alternative formulation would be one that encourages the discovery of another peer in the same end-network, but relaxes the constraints if such a peer cannot be found.

Backing up however, we note that the Meridian simulation results verify our earlier argument: It is hard to find the closest peer in clusters where the clusters have a large number of end-networks and the end-networks are all at about the same latencies to the respective cluster-hubs.

5.5 Mechanisms to Handle Clustering Effect

The previous sections argued how it would be hard to find the exact-closest peer in large P2P systems by examining inter-peer latencies alone. We next outline three basic approaches that try to solve the problem by incorporating additional information while finding the nearest peer. At the end of the section, we give a preliminary evaluation of the easiest to deploy of these approaches.

The first approach consists of a simple expanding search within each end-network using IP multicast; this search is aimed at finding other peers in the end-network. This technique has been suggested in previous work (e.g., to help find the nearest server [40], and to find existing peers in a P2P system to help bootstrap a new peer [86]). This approach however assumes that IP multicast is enabled within each end-network and that messages multicast from one host inside the end-network is capable of reaching any other host in the end-network; the latter assumption may often be invalid in large end-networks that are themselves composed of multiple LANs or VLANs.

The second approach uses a central server inside each end-network that tracks all peers inside the end-network that are currently in the P2P system. This server could conceivably be used to track membership in multiple P2P systems. The concern with this approach, aside from the obvious one regarding its centralized nature, is that it needs a sufficiently large number of peers within each end-network to justify the setup of the membership tracking server.

The third approach needs no explicit support from the network and can be implemented in a completely decentralized fashion. This approach uses hints to the actual *location* of a newly entering peer to help find its closest peer. The two hints we consider here are (i) The new peer's IP address, and (ii) The peer's *Upstream Connectivity List (UCL)*, i.e., the list of routers that are at a fixed number of hops (say 5) or closer from the peer, where peers would determine their UCLs by running traceroutes to a few different locations in the Internet. The intuition here is that two peers that have matching IP address prefixes or similar UCLs are likely to be close to each other.

We note that IP address prefixes and upstream routers have both been suggested as hints to proximity in previous work. CoralCDN [29] uses upstream routers to map clients to nearby servers, and to find latency-sensitive paths in an overlay. Freedman et al [28] note that IP addresses that share the same prefix are more likely to be in the same geographic location, and OASIS [30] uses IP address prefixes to again map clients to nearby servers. In this chapter however, we propose the use of UCLs and IP prefixes specifically to find the nearest peer, especially where the nearest peers share the same extended LAN.

The third approach requires a key-value mapping infrastructure to help peers find other peers with similar IP addresses or UCLs. In the UCL-based

heuristic, a mapping is created for each upstream router and peers that have the router in their UCLs: the key here is the IP address of the upstream router, and the value the IP addresses of the peers that have the router in their UCLs. When a new peer enters the system, it obtains its UCL, and uses the key-value map to retrieve IP addresses of all peers that it shares upstream routers with. The new peer can now actively probe the retrieved addresses to find the closest among them. The new peer inserts its own mapping once it joins the system. This approach ensures that peers that share a close upstream router would be able to find one another, provided that the IP address of the router is visible to the peers.

The IP-prefix based approach is similar to the above, except for the fact that the key used to store the mapping is a fixed-length prefix (e.g., the /24 prefix) of the peer's IP address.

The participant peers can themselves host the key-value maps required above, using one of several distributed hash table (DHT) designs available (Chord [99], CAN [80], Pastry [86], etc.). Many DHTs assume that keys are uniformly distributed, which may not be the case with IP addresses. In such scenarios, the IP addresses can be hashed to compute the keys to use in the system.

The hints used in the third approach, namely the UCL and the IP prefix, also have an additional (related) application beyond finding the nearest peer: They may be used in proximity-address based systems like Vivaldi and PIC [19, 17]. In these cases, the UCL (or the IP prefix) is added as an extension of the otherwise latency-based proximity address. When comparing two such composite addresses, if the UCL indicates that the nodes share an upstream router, then

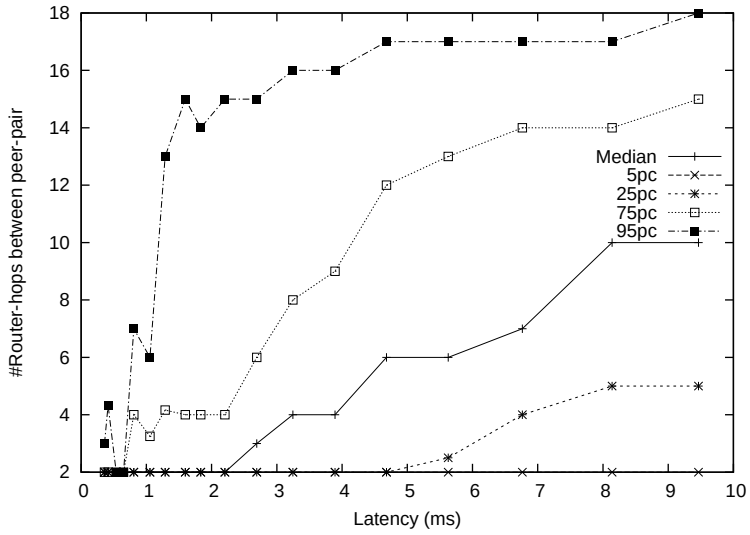


Figure 5.10: Inter-peer router hop-length as a function of inter-peer latency, for the UCL-based approach. The number of routers to be tracked in order to discover peers that are at a given latency range is equal to half the corresponding hop-length value.

the nodes are considered to be close together and the proximity address may be ignored. If the two nodes do not share an upstream router, then the UCL is ignored.

Note that the three approaches listed above would be used in conjunction with existing near-peer finding algorithms (and with one another) to obtain maximum accuracy in finding the nearest peer. The third approach (based on UCL or IP-prefix) however has the advantage of being able to be deployed in a decentralized fashion, and without need for extra network support.

We now give preliminary evaluations of both the UCL and IP-prefix based heuristics, using the Azureus peer-set from earlier (Section 5.3.2, [58, 69]). Our aim here is to investigate whether the heuristics are successful in finding nearby peers, and what the associated overheads are. We assume a perfect key-value map here for both approaches.

From the original peer-set, we retain those peers that responded with a valid latency to either a TCP ping or a traceroute (about 23000 peers). We track the latencies along traceroutes from the Planetlab vantage points to the different peers to get an approximate adjacency matrix: the matrix includes the Azureus peers and the routers along the traceroutes that responded with valid latencies, and tracks the latencies between the different routers and those between the routers and the Azureus peers. We run the Dijkstra algorithm over this adjacency matrix to obtain a set of closest peers for each peer, and show results for peer-pairs that are closer than 10 ms to each other – there are about 2400 peers that are within 10 ms to at least one other peer.

We present results for the UCL approach in Figure 5.10: it plots the router hop-lengths between close peer-pairs against the latencies between them. This plot is a "binned" scatter-plot of inter-peer hop-lengths versus inter-peer latencies, where sample points from nearby latencies are grouped into a single bin (similar to Figure 5.4). Note here that if all peers tracked upstream routers n hops away from them, they would be able to discover all peers $2n$ hops away, via the key-value map. So the fact that the bin at 3.9 ms has a median hop-length of 4 means that, in the median case, peers that make up the pairs in this bin would be able to discover each other (i.e., the other peers in the pairs) if each peer tracks its 2 closest upstream routers.

The figure shows that the UCL-based approach is indeed promising. The inter-peer hop-length grows with inter-peer latencies, implying that if the goal is to discover only very close peers, it can be achieved by having peers track only a modest number of routers: To discover peers closer than 5 ms, peers need to track 3 upstream routers each for a 50% success rate (the median case) and

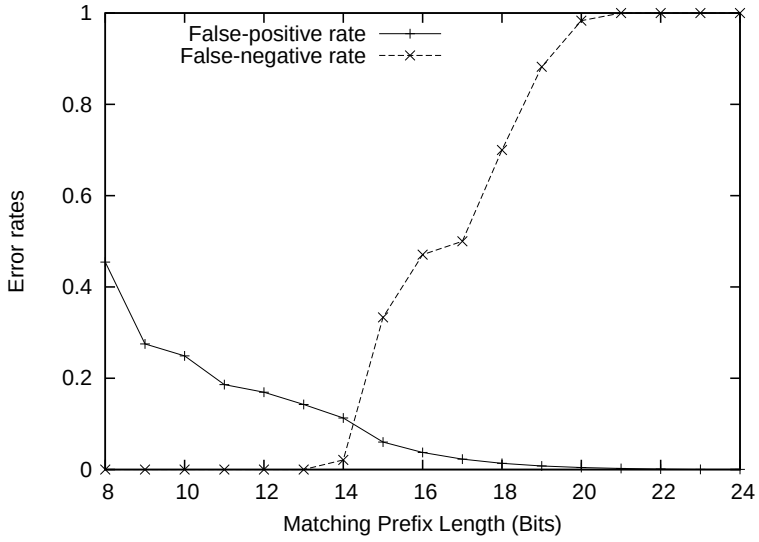


Figure 5.11: False-positive and false-negative rates with the IP-prefix based approach.

about 6 routers each for a 75% success rate. So this approach can be expected to perform very well when the closest peer is indeed very close-by in the general case. This also includes cases where the closest peer is in the same end-network.

On the other hand, the growing hop-length with latency has negative implications for this approach if the closest peer happens to be significantly farther away, and the goal is to still discover that closest peer. In such scenarios, we suggest coupling the above approach with traditional nearest-peer algorithms.

Figure 5.11 shows results for the IP-prefix based heuristic. It shows the median false-positive and false-negative rates incurred by the approach as a function of different prefix-lengths. For each peer, we compute the false-positive rate as the ratio of the number of peers that share the same IP prefix as the given peer, but are more than 10 ms away from the peer, to the total number of peers that are more than 10 ms away from the peer. Similarly, the false-negative rate is the ratio of the number of peers with a different IP prefix, but are closer than 10

ms to the peer, to the total number of peers that are closer than 10 ms to the peer. We again estimate the latency between peers as the latency along the shortest path in the traceroute-generated graph. The population-size here is about 2400 (as mentioned before). It is desirable of course that both the false-positive and false-negative rates are low: If the false-positive rate is high, a lot of effort is expended in further probing the nodes returned by the heuristic to actually find the few nodes that are close-by. Similarly, if the false-negative rate is high, a large proportion of the peers that are actually close-by are never found.

Figure 5.11 shows, as expected, that the false-positive rate falls with more fine-grained (longer) prefixes, whereas the false-negative rate increases with longer prefixes. Unfortunately, there is no clear "sweet-spot" here: With a prefix-length of 14 bits or shorter, the false-positive rate is greater than 0.1, so at least about 250 peers need to be further probed to identify those peers that are actually close. And with larger prefix-lengths, more and more close-by peers are ignored.

The UCL-based approach, on the other hand, is not vulnerable to the above false-positive problem: In the mapping of upstream routers to end-host IP addresses, we could also embed information about the latency between the routers and the end-hosts. Two peers that share upstream routers can now form a rough estimate of their latency to each other as the sum of their latencies to the closest common router. Thus peers can discard, without further probing, other peers that are estimated to be too far away.

Since we do not control the end-hosts used in the measurements here, we are unable to empirically observe occurrences of false-negatives with the UCL approach. In practice, this will depend on the completeness of the UCL map that

peers can generate: the more complete the maps are, the less is the possibility of false-negatives.

5.6 Conclusions and Future Work

In this chapter, we identified the clustering condition, and showed that it makes it expensive for latency-only based proximity methods to find extreme-nearby (same campus network) peers in the Internet. We performed large-scale measurements over the Internet to show, with reasonable confidence, that the clustering condition does occur in real settings, and used analytical arguments and simulations to show that nearest-peer finding algorithms suffer under the condition. We listed different approaches to overcome this issue, and showed that one of them was quite promising. Overall, this chapter showed that developers of latency-sensitive P2P applications need to be mindful of this factor when deploying their systems, and should employ additional mechanisms like those suggested in this chapter when finding extreme-nearby peers is important.

An interesting line of future work is to determine the exact extent of occurrence of the clustering condition in particular deployed P2P systems. Doing so would however require explicit cooperation from the individual peers. Another supplementary piece of future work is to more extensively evaluate all the different mechanisms proposed in the chapter to handle the clustering condition.

5.7 Acknowledgments

We thank Aaron Sidford for help in generating the DNS measurement data. We thank Hitesh Ballani for sharing the DNS server list, Jonathon Ledlie for sharing the Azureus client IP list, and Bernard Wong for sharing the Meridian latency dataset and the Meridian simulator. Finally, we thank Aleksandrs Slivkins and Jon Kleinberg for helpful discussions about growth-constrained metrics and doubling metrics.

CHAPTER 6

SUMMARY, CONCLUSIONS AND FUTURE WORK

P2P applications have been immensely popular in the recent past, and account for a major portion of the total Internet traffic today. While the usage of P2P applications has been popular with individual users for long, more recently, enterprises are deploying P2P technologies for load-balancing within the enterprise, or to offload bandwidth and processing costs.

In this thesis, we recognized and devised solutions to three common problems that occur across different P2P systems: (i) heterogeneous random graph construction, (ii) heterogeneous random peer selection, and (iii) nearest neighbor discovery. The first two come under the common umbrella of load-balancing in heterogeneous unstructured networks: here capacities to support load differ between the different members, and the application load is to be distributed in accordance with members' capacities to support it. The above problems occur in settings like file-sharing, overlay multicast, online games, gossip-based protocols, proximity-based systems, and others. Developers of these diverse applications can therefore reuse our solutions, instead of having to repeatedly solve the same problems.

We studied various unstructured approaches to do heterogeneous graph construction and peer selection in Chapter 3, and identified the *Swaplinks* algorithm as the most attractive heterogeneous random graph construction algorithm. Swaplinks builds robust graphs where node degrees are close to their desired degrees, is efficient and scalable, and is virtually free of tuning knobs, making it very practical to deploy. Moreover, the Swaplinks graph provides a good base to perform random peer selection: simple random walks (e.g., On-

lyInLinks, TotalInvProb) on top of the Swaplinks graph result in the desired selection, where frequency of node-selection is in proportion to node-capacities.

We compared unstructured and structured approaches to perform heterogeneous peer selection in Chapter 4: We used Swaplinks-based selection as the candidate unstructured approach, and *KRB*, Karger and Ruhl's algorithms adapted to a heterogeneous setting, as the candidate structured approach. We found that Swaplinks in general is a better selection approach: KRB struggles to maintain the desired selection quality under high churn, and is much harder to configure to extract the optimal performance. In general, we believe that unstructured approaches are more appropriate than structured approaches for peer-selection, because peer-selection is inherently an unstructured problem.

Finally, in Chapter 5, we considered the problem of discovering the latency-wise closest peer in P2P systems, especially where the closest peer is in the same extended LAN or campus network. We identified and demonstrated (using measurements) the *clustering condition* that appears in the Internet in this setting: under the clustering condition, many different networks of peers all appear to be at about the same latency from another, making it hard to discover peers from one's own network from the entire population of peers. We showed, using analysis and simulations, that existing nearest-neighbor approaches are not practical to use in this setting. Existing approaches only use (measured or predicted) inter-peer latencies to discover the nearest peer, and make various assumptions about the distribution of inter-peer latencies that fail to hold under the clustering condition. We proposed multiple solutions that take into account the network topology, in addition to inter-peer latencies. Preliminary evaluations show that one of these, which tracks the upstream routers of each peer, is

very promising.

6.1 Limitations, Future Enhancements, and Open Problems

The foremost of the limitations of many of the approaches proposed in this thesis is that they are vulnerable to malicious or misbehaving nodes. In the unstructured graph-building mechanisms we studied, a malicious node can accumulate links to a majority of the nodes in the network if it so chooses, without having to set its desired degree to an inordinately large value. We need to explore simple mechanisms to prevent this. In our upstream-router based solution to the nearest-peer discovery problem, a malicious node can spuriously mark itself close to many different upstream routers, implying that it would then be close to all peers that have the routers as their own upstream routers as well. Here however, the problem is not as serious, since probing of potential nearest-peers before accepting them as such is a simple solution to the attack.

In terms of enhancements, we need to experiment further with smart-pinging in Swaplinks, in order to study the trade-off between message loads when there is no churn versus those when there is churn, and to insure that it does not suffer from false negatives. As far as our nearest-neighbor work is concerned, we need to determine the exact extent of occurrence of the clustering condition in particular deployed P2P systems, so as to precisely gauge the effect of the clustering condition. Doing so would however require explicit cooperation from the individual peers. We also need to more extensively evaluate the different mechanisms proposed in Chapter 5 to handle the clustering condition.

The nearest-peer approaches we propose are intended for the special but

important case where the actual nearest peer is in the same extended LAN or campus network as the node that is trying to find the nearest peer. Accordingly, our solutions are not applicable when the latency between nearest peers is large (say more than 10 ms). Since previously proposed nearest-peer solutions target this latter setting, we expect our solutions to be used in conjunction with previous solutions in a real deployment.

Our solutions to the problems of heterogeneous unstructured load-balancing and nearest-peer discovery individually address the problems they are intended to solve. There are however many settings where it would be beneficial to select nodes based on both capacity and proximity. While some simple solutions are available, e.g., use heterogeneous random selection to pick a few nodes, and then pick the closest among them, or vice-versa, more research is needed to find the best solution.

6.2 Offshoots from Thesis Research

Comparison of previous nearest-peer algorithms: Numerous algorithms have been previously proposed to discover the nearest peer in P2P systems. As noted before, the previous algorithms target settings where the nearest peers are not too close to each other, and where inter-peer latencies are "well-distributed".[1] In spite of the wealth of the previous proposals, and their diversity, there has yet been no research aimed at determining the best scheme (i.e., the most efficient) among these. It would be a worthwhile piece of future work to perform a thorough performance comparison of the different schemes.

[1] See Section 2.3 and Chapter 5 for previous assumptions on the latency-space.

Discovering high-bandwidth peers: Bandwidth between peers, in addition to the latency, forms an important parameter of interest when picking neighbors in P2P systems (e.g., file-sharing). As compared to latency however, discovering peers with large bandwidths to a given peer is a harder problem to solve. Fortunately, the latency between two peers and the bandwidth capacity are likely to be strongly and inversely correlated at small latencies (say less than 5 ms). This is because peers this close should also be topologically close to each other, with very few routers and links on the path between them. There has been prior research examining the correlation of the bandwidth achieved by TCP with end-to-end RTT. However, there is no known research that looks at the correlation of the bandwidth *capacity* and latency when the two endpoints are close to each other in an ISP's access topology. If the above conjecture is borne out, finding the nearest peer is doubly important, because a peer's nearest peer is also likely to have a high bandwidth to the peer. Verifying this conjecture will require an extensive measurement over Internet end-hosts.

Decentralized Enterprise Load-Balancing: Enterprises have deployed thousands of (possibly even hundreds of thousands of) servers in single physical locations to efficiently handle user requests. These setups typically use a hierarchical load-balancing architecture. While the hierarchical design offers performance benefits, it is possible that it presents a bottleneck in scaling to even larger networks: the higher in the hierarchy a component is, the greater is the disruption caused by its failure, thus affecting the availability of the overall service. Recently, P2P architectures have been adopted to allay this concern: e.g., Amazon's Dynamo storage infrastructure uses a DHT-based design.

It would be interesting to see how well a Swaplinks-style *unstructured* P2P

load-balancing scheme would perform in these settings, especially where the servers are stateless: a prime example would be a grid-computing application with minimal network and database interactions. In comparison to a DHT-based scheme, an unstructured scheme places fewer constraints on be the inter-server connectivity graph, so should be more robust and scalable. Challenges here include overcoming a possibly increased latency in finding a random server, and a possibly reduced overall quality of load-balancing due to the lack of global information at every server.

BIBLIOGRAPHY

[1] Lada A. Adamic, Rajan M. Lukose, Bernardo Huberman, and Amit R. Puniyani. Search in Power-Law Networks. *Phys. Rev. E*, 64(046135), 2001.

[2] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: scaling file servers via cooperative caching. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 129–142, Berkeley, CA, USA, 2005. USENIX Association.

[3] Hitesh Ballani, Paul Francis, and Sylvia Ratnasamy. A Measurement-based Deployment Proposal for IP Anycast. In *Proc. IMC*, 2006.

[4] Bamboo DHT – Download. `http://bamboo-dht.org/download.html`, Accessed June 2008.

[5] Suman Banerjee, Christopher Kommareddy, and Bobby Bhattacharjee. Scalable Peer Finding on the Internet. In *Proc. Global Internet Symposium, Globecom*, 2002.

[6] Ziv Bar-Yossef, Alexander Berg, Steve Chien, Jittat Fakcharoenphol, and Dror Weitz. Approximating Aggregate Queries about Web Pages via Random Walks. In *Proc. VLDB*, 2000.

[7] Marcin Bienkowski, Miroslaw Korzeniowski, and Friedhelm Meyer auf der Heide. Dynamic Load Balancing in Distributed Hash Tables. In *IPTPS*, pages 217–225, 2005.

[8] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, 1999.

[9] Bittorrent, `http://www.bittorrent.com/`.

[10] John W. Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In *IPTPS*, pages 80–87, 2003.

[11] Miguel Castro, Manuel Costa, and Antony Rowstron. Should we build Gnutella on a structured overlay? In *Proc. HotNets-II*, 2003.

[12] Miguel Castro, Manuel Costa, and Antony Rowstron. Debunking Some Myths About Structured and Unstructured Overlays. In *Proc. NSDI*, 2005.

[13] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Proximity neighbor selection in tree-based structured peer-to-peer overlays. In *Technical report MSR-TR-2003-52*, 2003.

[14] Yatin Chawathe, Sylvia Ratnasamy, Lee Breslau, Nick Lanham, and Scott Shenker. Making Gnutella-like P2P systems scalable. In *Proc. ACM SIGCOMM*, 2003.

[15] Yan Chen, Khian Hao Lim, Randy H. Katz, and Chris Overton. On the Stability of Network Distance Estimation. *SIGMETRICS Perform. Eval. Rev.*, 30(2):21–30, 2002.

[16] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *LNCS*, pages 46–66. Springer-Verlag, 2001.

[17] Manuel Costa, Miguel Castro, Antony I. T. Rowstron, and Peter B. Key. PIC: Practical Internet Coordinates for Distance Estimation. In *Proc. ICDCS*, 2004.

[18] I Csiszár. Information theoretic methods in probability and statistics. In *IEEE Information Theory Society Newsletter 48*, 1998.

[19] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A Decentralized Network Coordinate System. In *Proc. ACM SIGCOMM*, 2004.

[20] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proc. ACM SOSP*, 2001.

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[22] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proc. PODC*, 1987.

[23] Ellacoya Data Shows Web Traffic Overtakes Peer-to-Peer (P2P) as Largest

Percentage of Bandwidth on the Network. `www.ellacoya.com/news/pdf/2007/NXTcommEllacoyaMediaAlert.pdf`, Accessed June 2008.

[24] P. Eugster, S. Handurukande, R. Guerraoui, A. Kermarrec, and P. Kouznetsov. Lightweight probabilistic broadcast. In *Proceedings of The International Conference on Dependable Systems and Networks (DSN 2001)*, July 2001.

[25] Rodrigo Fonseca, Puneet Sharma, Sujata Banerjee, Sung-Ju Lee, and Sujoy Basu. Distributed Querying of Internet Distance Information. In *Proceedings of the 8th IEEE Global Internet Symposium, Miami, FL*, 2005.

[26] Paul Francis. Yoid: Extending the Internet Multicast Architecture. In *Unrefereed report*, 2000.

[27] Paul Francis, Sugih Jamin, Cheng Jin, Yixin Jin, Danny Raz, Yuval Shavitt, and Lixia Zhang. IDMaps: a global internet host distance estimation service. In *IEEE/ACM Trans. Netw. 9(5): 525-540*, 2001.

[28] Michael Freedman, Mythili Vutukuru, Nick Feamster, and Hari Balakrishnan. Geographic Locality of IP Prefixes. In *Proc. IMC*, 2005.

[29] Michael J. Freedman, Eric Freudenthal, and David Mazires. Democratizing Content Publication with Coral. In *Proc. NSDI*, 2004.

[30] Michael J. Freedman, Karthik Lakshminarayanan, and David Mazieres. OASIS: Anycast for any service. In *Proc. NSDI*, 2006.

[31] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulie. SCAMP: peer-to-peer lightweight membership service for large-scale group communication. In *Proc. 3rd Intnl. Wshop Networked Group Communication (NGC '01)*, pages 44–55, 2001.

[32] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massouli. Peer-to-Peer Membership Management for Gossip-Based Protocols. In *IEEE Trans. Computers 52(2)*, pages 139–149, 2003.

[33] Christos Gkantsidis, Milena Mihail, and Amin Saberi. Random Walks in Peer-to-Peer Networks. In *Proc. IEEE Infocom*, 2004.

[34] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard

Karp, and Ion Stoica. Load balancing in dynamic structured P2P systems. In *Proc. IEEE Infocom*, 2004.

[35] P. Brighten Godfrey and Ion Stoica. Heterogeneity and load balance in distributed hash tables. In *Proc. IEEE Infocom*, 2005.

[36] David M. Goldschlag, Michael G. Reed, and Paul F. Syverson. Hiding routing information. In *Proceedings of the First International Workshop on Information Hiding*, pages 137–150, London, UK, 1996. Springer-Verlag.

[37] Saikat Guha and Paul Francis. Characterization and Measurement of TCP Traversal through NATs and Firewalls. In *Proc. IMC*, 2005.

[38] Krishna P. Gummadi, Richard J. Dunn, Stefan Saroiu, Steven D. Gribble, Henry M. Levy, and John Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proc. ACM SOSP*, 2003.

[39] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: Estimating Latency between Arbitrary Internet End Hosts. In *Proc. ACM SIGCOMM*, 2002.

[40] James D. Guyton and Michael F. Schwartz. Locating Nearby Copies of Replicated Internet Servers. In *Proc. ACM SIGCOMM*, 1995.

[41] W. Hastings and Monte Carlo. Sampling Methods Using Markov Chains and Their Applications. *Biometrika*, 57, 1970.

[42] Kirsten Hildrum, John D. Kubiatowicz, Satish Rao, and Ben Y. Zhao. Distributed Object Location in a Dynamic Network. In *Proc. ACM SPAA*, 2002.

[43] S.M. Hotz. Routing information organization to support scalable interdomain routing with heterogeneous path requirements. In *Ph.D. Thesis, University of Southern California*, 1994.

[44] Yang hua Chu, Aditya Ganjam, T. S. Eugene Ng, Sanjay G. Rao, Kunwadee Sripanidkulchai, Jibin Zhan, and Hui Zhang. Early deployment experience with an overlay based internet broadcasting system. In *Proc. Usenix Annual Technical Conference*, 2004.

[45] Yang hua Chu, Sanjay G. Rao, and Hui Zhang. A Case for End System Multicast. In *Proc. ACM SIGMETRICS*, 2000.

[46] Ipoque Internet Study 2007. http://www.ipoque.com/news_&_events/internet_studies/internet_study_2007, Accessed June 2008.

[47] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3):8, 2007.

[48] D. Karger and M. Ruhl. Finding Nearest Neighbors in Growth-restricted Metrics. In *Proc. ACM STOC*, 2002.

[49] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. ACM STOC*, 1997.

[50] David R. Karger and Matthias Ruhl. New Algorithms for Load Balancing in Peer-to-Peer Systems. In *IRIS Student Workshop*, 2003.

[51] David R. Karger and Matthias Ruhl. Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems. In *Proc. IPTPS*, 2004.

[52] Kazaa Supernode FAQ. http://www.kazaa.com/us/help/faq/supernodes.htm, Accessed June 2008.

[53] T. Klingberg and R. Manfredi. Gnutella 0.6 RFC, 2002.

[54] Christopher Kommareddy, Narendar Shankar, and Bobby Bhattacharjee. Finding Close Friends on the Internet. In *Proc. IEEE ICNP*, 2001.

[55] Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using Random Subsets to Build Scalable Network Services. In *Proc. USITS*, 2003.

[56] Dejan Kostic, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High Bandwidth Data Dissemination Using an Overlay Mesh. In *Proc. ACM SOSP*, 2003.

[57] Ching Law and Kai-Yeung Siu. Distributed construction of random expander networks. In *Proc. IEEE Infocom*, 2003.

[58] Jonathan Ledlie, Paul Gardner, and Margo Seltzer. Network Coordinates in the Wild. In *Proc. NSDI*, 2007.

[59] Jonathan Ledlie and Margo Seltzer. Distributed, Secure Load Balancing with Skew, Heterogeneity, and Churn. In *Proc. IEEE Infocom*, 2005.

[60] Jinyang Li, Jeremy Stribling, Robert Morris, and M. Frans Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proc. NSDI*, 2005.

[61] Jian Liang, Rakesh Kumar, and Keith W. Ross. The Kazaa Overlay: A Measurement Study. In *Computer Networks (Special Issue on Overlays)*, 2005.

[62] Jin Liang and Klara Nahrstedt. RandPeer: Membership Management for QoS Sensitive Peer-to-Peer Applications. In *INFOCOM*, 2006.

[63] Hyuk Lim, Jennifer C. Hou, and Chong-Ho Choi. Constructing Internet Coordinate System Based on Delay Measurement. In *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 129–142, New York, NY, USA, 2003. ACM.

[64] Qin Lv, Pei Cao, Edith Cohen, Kai Li, and Scott Shenker. Search and replication in unstructured peer-to-peer networks. In *In ICS'02*, 2002.

[65] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS*, pages 53–65, 2002.

[66] Roie Melamed and Idit Keidar. Araneola: A Scalable Reliable Multicast System for Dynamic Environments. In *Proc. NCA 2004*, 2004.

[67] Meridian: Lightweight Positioning. `http://www.cs.cornell.edu/People/egs/meridian/`, Accessed June 2008.

[68] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, and Augusta H. Teller. Equations of State Calculations by Fast Computing Machines. In *Journal of Chemical Physics*, 1953.

[69] Network Coordinate Research at Harvard. `http://www.eecs.harvard.edu/~syrah/nc/`, Accessed June 2008.

[70] T. S. Eugene Ng and Hui Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *Proc. IEEE Infocom*, 2002.

[71] T. S. Eugene Ng and Hui Zhang. A Network Positioning System for the Internet. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2004. USENIX Association.

[72] Vinay Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander E. Mohr. Chainsaw: Eliminating Trees from Overlay Multicast. In *Proc. IPTPS*, 2005.

[73] C. Partridge, T. Mendez, and W. Milliken. RFC 1546 – Host Anycasting Service, 1993.

[74] Marcelo Pias, Jon Crowcroft, Steve R. Wilbur, Tim Harris, and Saleem N. Bhatti. Lighthouses for Scalable Distributed Location. In *IPTPS*, pages 278–291, 2003.

[75] Planetlab. http://www.planet-lab.org.

[76] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[77] Peter Quax, Patrick Monsieurs, Wim Lamotte, Danny De Vleeschauwer, and Natalie Degrande. Objective and Subjective Evaluation of the Influence of Small Amounts of Delay and Jitter on a Recent First Person Shooter Game. In *Proc. ACM SIGCOMM workshop on Network and system support for games*, 2004.

[78] Van Renesse R., Minsky Y., and Hayden M. A gossip-style failure detection service. In *Middleware*, page 5570, 1998.

[79] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard M. Karp, and Ion Stoica. Load Balancing in Structured P2P Systems. In *IPTPS*, 2003.

[80] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM 2001*, 2001.

[81] Sylvia Ratnasamy, Mark Handley, Richard M. Karp, and Scott Shenker. Application-level multicast using content-addressable networks. In *NGC*

'01: Proceedings of the Third International COST264 Workshop on Networked Group Communication, pages 14–29, London, UK, 2001. Springer-Verlag.

[82] Sylvia Ratnasamy, Mark Handley, Richard M. Karp, and Scott Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proc. IEEE Infocom*, 2002.

[83] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *Proc. Usenix Annual Technical Conference*, 2004.

[84] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: a public DHT service and its uses. *SIGCOMM Comput. Commun. Rev.*, 35(4):73–84, 2005.

[85] Matei Ripeanu and Ian Foster. Mapping Gnutella Network: Macroscopic Properties of Large-Scale Peer-to-Peer Systems. In *Proc. 1st International Workshop on Peer-to-Peer Systems (IPTPS02)*, Cambridge, MA, March 2002.

[86] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware*, 2001.

[87] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. ACM SOSP*, 2001.

[88] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proc. NGC*, 2001.

[89] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. MMCN*, 2002.

[90] Subhabrata Sen and Jia Wang. Analyzing Peer-To-Peer Traffic Across Large Networks. In *Second Annual ACM Internet Measurement Workshop*, 2002.

[91] Puneet Sharma, Zhichen Xu, Sujata Banerjee, and Sung-Ju Lee. Estimating network proximity and latency. *SIGCOMM Comput. Commun. Rev.*, 36(3):39–50, 2006.

[92] Yuval Shavitt, Xiaodong Sun, Avishai Wool, and Bülent Yener. Computing the unmeasured: an algebraic approach to Internet mapping. *IEEE Journal on Selected Areas in Communications*, 22(1):67–78, 2004.

[93] Yuval Shavitt and Tomer Tankel. Big-Bang Simulation for embedding network distances in Euclidean space. In *INFOCOM*, 2003.

[94] Haiying Shen and Cheng-Zhong Xu. Locality-Aware and Churn-Resilient Load-Balancing Algorithms in Structured Peer-to-Peer Networks. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):849–862, 2007.

[95] Ryan Singel. Internet Mysteries: How Much File Sharing Traffic Travels the Net? – Update. `http://blog.wired.com/27bstroke6/2008/05/how-much-file-s.html`, Accessed June 2008.

[96] Aleksandrs Slivkins. Embedding, Distance Estimation and Object Location in Networks. In *Ph.D. Thesis, Cornell University*, 2006.

[97] Neil Spring, David Wetherall, , and Tom Anderson. Scriptroute: A Public Internet Measurement Facility. In *Proc. USITS*, 2003.

[98] Tyler Steele, Vivek Vishnumurthy, and Paul Francis. A Parameter-Free Load Balancing Mechanism For P2P Networks. *IPTPS*, 2008.

[99] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. ACM SIGCOMM*, 2001.

[100] Daniel Stutzbach, Reza Rejaie, Nick G. Duffield, Subhabrata Sen, and Walter Willinger. On unbiased sampling for unstructured peer-to-peer networks. In *Internet Measurement Conference*, pages 27–40, 2006.

[101] Qixiang Sun and Daniel Sturman. A Gossip-Based Reliable Multicast for Large-Scale High-Throughput Applications. In *International Conference on Dependable Systems and Networks (DSN)*, 2000.

[102] Liying Tang and Mark Crovella. Virtual Landmarks for the Internet. In *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 143–152, New York, NY, USA, 2003. ACM.

[103] Wolfgang Theilmann and Kurt Rothermel. Dynamic Distance Maps of the Internet. In *INFOCOM*, pages 275–284, 2000.

[104] Vidhyashankar Venkataraman, Kaouru Yoshida, and Paul Francis. Chunkyspread: Heterogeneous Unstructured Tree-based Peer-to-Peer Multicast. In *Proc. ICNP*, 2006.

[105] Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gün Sirer. KARMA: A Secure Economic Framework for P2P Resource Sharing. In *Workshop on Economics of Peer-to-Peer Systems, Berkeley, CA, USA*, June 2003.

[106] Vivek Vishnumurthy and Paul Francis. On Heterogeneous Overlay Construction and Random Node Selection in Unstructured P2P Networks. In *Proc. IEEE Infocom*, Barcelona, Spain, 2006.

[107] Vivek Vishnumurthy and Paul Francis. A Comparison of Structured and Unstructured P2P Approaches to Heterogeneous Random Peer Selection. In *Proc. Usenix Annual Technical Conference*, 2007.

[108] Marcel Waldvogel and Roberto Rinaldi. Efficient Topology-Aware Overlay Network. In *Computer Communication Review 33(1): 101-106*, 2003.

[109] Li wei Lehman and Steven Lerman. PCoord: Network Position Estimation Using Peer-to-Peer Measurements. In *Proc. NCA*, 2004.

[110] Drew Wilson. CacheLogic Study – P2P is Changing. `http://www.slyck.com/story914_CacheLogic_Study_P2P_is_Changing`, Accessed June 2008.

[111] Bernard Wong, Aleksandrs Slivkins, and Emin Gün Sirer. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In *Proc. ACM SIGCOMM*, 2005.

[112] Ellen W. Zegura, Kenneth L. Calvert, and Samrat Bhattacharjee. How to Model an Internetwork. In *Proc. IEEE Infocom*, 1996.

[113] Ming Zhong, Kai Shen, and Joel Seiferas. The Convergence-Guaranteed Random Walk and Its Applications in Peer-to-Peer Networks. *IEEE Transactions on Computers*, 57(5):619–633, 2008.

[114] Yingwu Zhu and Yiming Hu. Efficient, Proximity-Aware Load Balancing for DHT-Based P2P Systems. *IEEE Trans. Parallel Distrib. Syst.*, 16(4):349–361, 2005.

[115] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. ACM NOSSDAV*, 2001.